# ISP Customer Churn Machine Learning Analysis

*Project Report (Group 11):* Max Enabnit, Greg Fagan, Tommy Jaeger

# Introduction:

## Problem Background

For our project, we have chosen to analyze a dataset containing 3 months of data on customer churn rates and other characteristics for Internet Service Providers. When looking at the dataset, we began to wonder what variables of customer experience causes customers to churn, canceling their subscriptions with their current provider and taking their business to another. Valuations of customers and understanding why they leave their current providers is crucial to better business understanding and customer service, which largely impact business success among internet service providers.

## Data Understanding

We retrieved our dataset from Kaggle, where our dataset houses internet service companies' customer tenure data with 10 columns and 72,000+ unique customers. This dataset spans over 3 months' time. Some datatypes that can be found in the dataset are:

**Integer**: ID (Unique Customer ID), bill_avg(last 3 months bill average per customer), service_failure_count(customer call count to call center for service failure over 3 months), download_over_limit(For customers who exceed the limit, there should be additional charges. However, the dataset does not indicate any payments for exceeding the limit)

**Double**: subscription_age (how long someone has been a customer), remaining_contract (how many years remain in customer contract, if null; customer doesn't have a contract, if cancelled they pay a penalty fee), download_avg (last 3 months of internet usage(GB)), upload_avg (last 3 months of upload avg (GB))

**Binary**: is_tv_subscriber (whether customer has tv subscription), is_movie_package_subscriber (whether customer has movie package subscription), churn (0 customer did not leave, 1 customer left internet service provider)

# Brainstorming Questions:

From looking at the data, four primary questions arose for our group:

1. Can a business predict the average lifetime value of a customer over their current subscription?
2. How do subscription age and internet usage specifically impact churn?
3. How can we segment customers and churn based on their usage patterns, subscription type and behavior?
4. What attributes have the largest impact on churn?

**Continue to Preprocess + EDA**

# Preprocessing:

Preprocessing our dataset had a couple steps to ensure everything ran smoothly for our modeling section. We will start by addressing null values, I first added our SQL table isp_df into a spark DataFame and counted all NULL values with this code block:

```
from pyspark.sql.functions import col, count, when

# Step 2: Count the NULL values for each column in the DataFrame

null_counts = isp_df.select([count(when(col(column).isNull(), 1)).alias(column) for column in isp_df.columns])

null_counts.display()
```

This gave us a resulting DataFrame that had the totals of NULL values in each column. This meant we could figure out which columns were problematic. There were 21,572 NULL values for remaining_contract and 381 NULL values for download_avg and upload_avg. To handle these NULL values we will create a new column called contract_status so that we can visualize and use this column in our modeling.

```
isp_df = isp_df.withColumn(

    "remaining_contract",

    when(col("remaining_contract").isNull(), -1).otherwise(col("remaining_contract"))

)

display(isp_df)
```

AND

```
isp_df = isp_df.withColumn(

    "contract_status",

    when(col("remaining_contract") == -1, "never_had_contract")

    .when(col("remaining_contract") == 0, "expired_contract")

    .otherwise("active_contract")

)

display(isp_df)
```

This will convert the column into -1 if it is null, this is because a customer can be a customer without a contract meaning this is null, or the customer never had a contract. Then the second block of code will turn that into 'never_had_contract' and then 0 would be 'expired_contract.'

```
isp_df.groupBy("contract_status").count().show()
```

Then we will group by contract_status, our new column to see counts of how our data is distributed:

**never_had_contract:** 21572

**expired_contract:** 16363

**active_contract:** 34339

Next, we will have to address this column and convert them from a categorical column 'contract_status' into a numeric column using StringIndexer. This needs to be implemented for the use of the machine learning in our later analysis.

```
from pyspark.ml.feature import StringIndexer

# Initialize StringIndexer

indexer = StringIndexer(inputCol="contract_status", outputCol="contract_status_index",
handleInvalid="skip")

# Fit and transform the data

isp_df = indexer.fit(isp_df).transform(isp_df)

# Show the result

isp_df.select("contract_status", "contract_status_index").show()
```

This will convert the following categorical features into numerical:

**expired_contract:** 2.0

**never_had_contract:** 1.0

**active_contract:** 0.0

I will then use these numerical features to one hot encode using this code block:

```
from pyspark.ml.feature import OneHotEncoder
```

```
# Step 2: OneHotEncoder for contract_status_index

encoder = OneHotEncoder(inputCol="contract_status_index",
outputCol="contract_status_onehot")

# Fit and transform the data

isp_df = encoder.fit(isp_df).transform(isp_df)

# Show the result

isp_df.select("contract_status", "contract_status_index", "contract_status_onehot").show()
```

This will one hot encode our data for model learning, it will look like:

**active_contract:** [1.0, 0.0]

**never_had_contract:** [0.0, 1.0]

**expired_contract:** [0.0, 0.0]

We will now focusing on removing missing rows for upload/download average values. We decided the best course of action is to remove these rows because all of these rows have a similarity; if download_avg is null then upload_avg is also null. It is also only 300 rows of missing data so it would be best to remove them.

```
isp_df = isp_df.na.drop(subset=["upload_avg", "download_avg"])
```

This will remove values and now we can move on to some summary statistics and EDA.

# Summary Statistics + EDA

**Running some summary statistics on our table we see the following:**

**Count(rows):** 71,893 rows

**Average tv_subscriber:** .85 or 85%

**Average movie_subscriber:** .33 or 33%

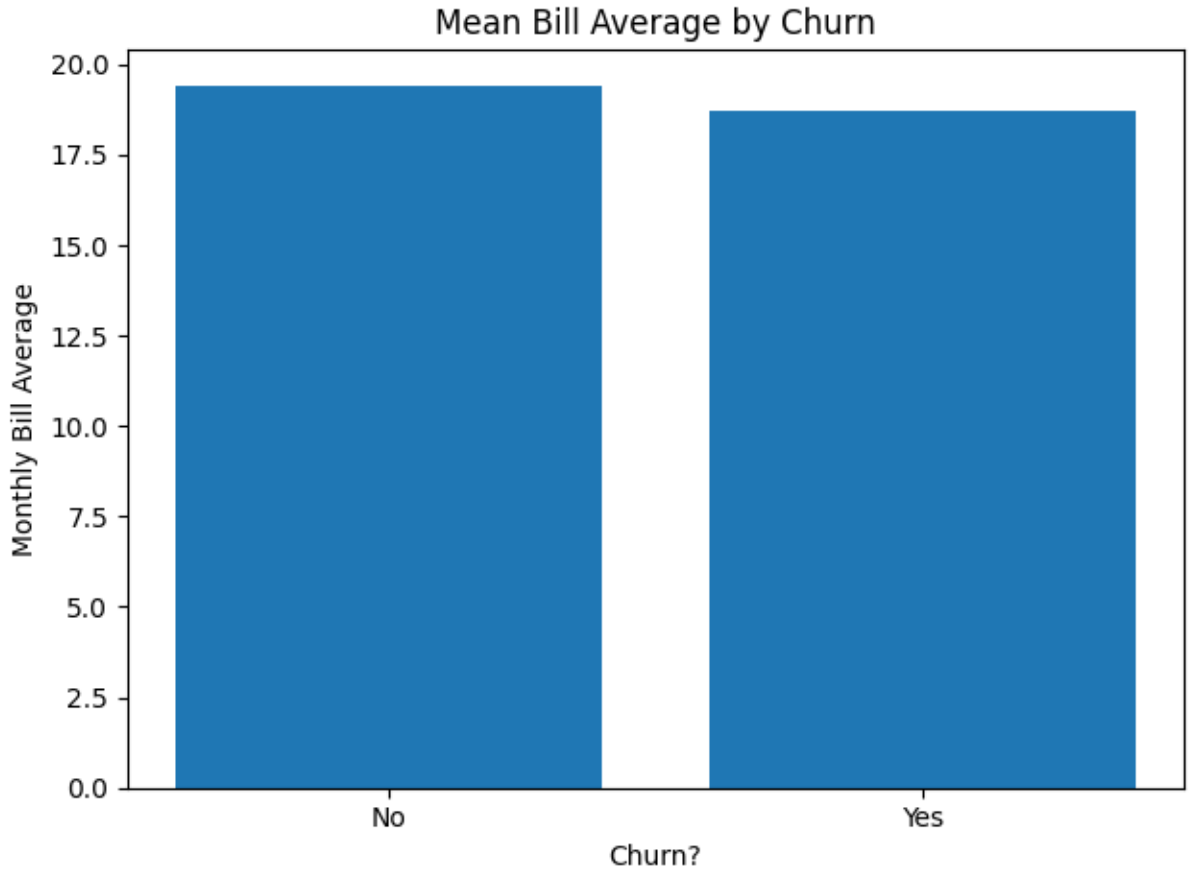
**Average subscription_age:** 2.45 years

**Average download_avg:** 43.68

**Highest download_avg:** 4415.2

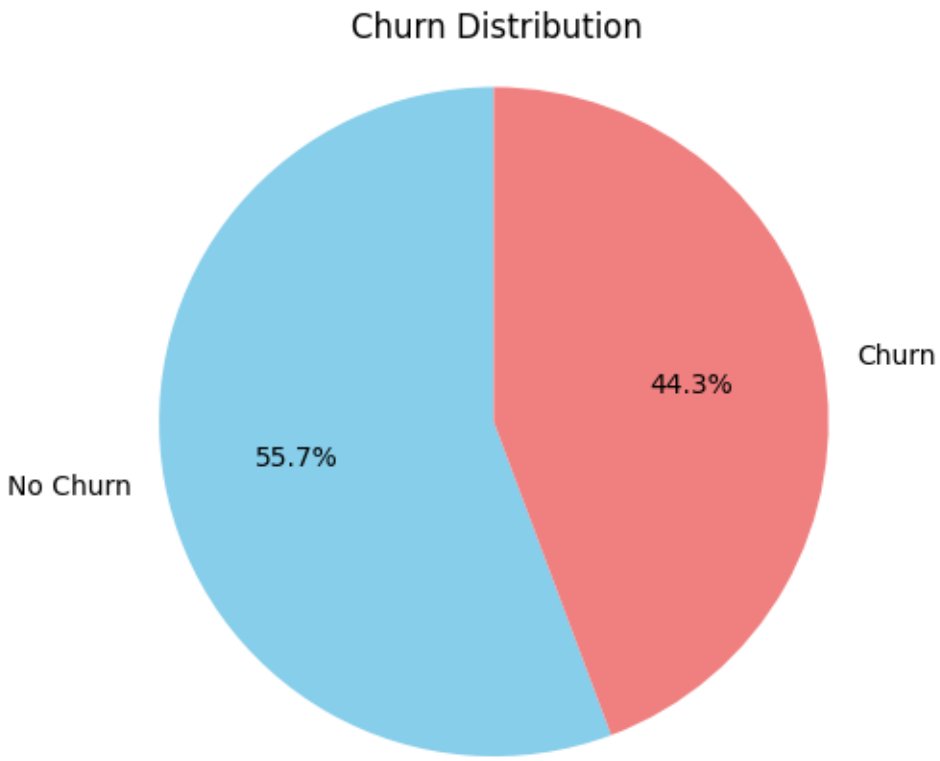
**Average churn rate:** .55 or 55%

**EDA 1: Average Monthly Bill by Churn Rate:**



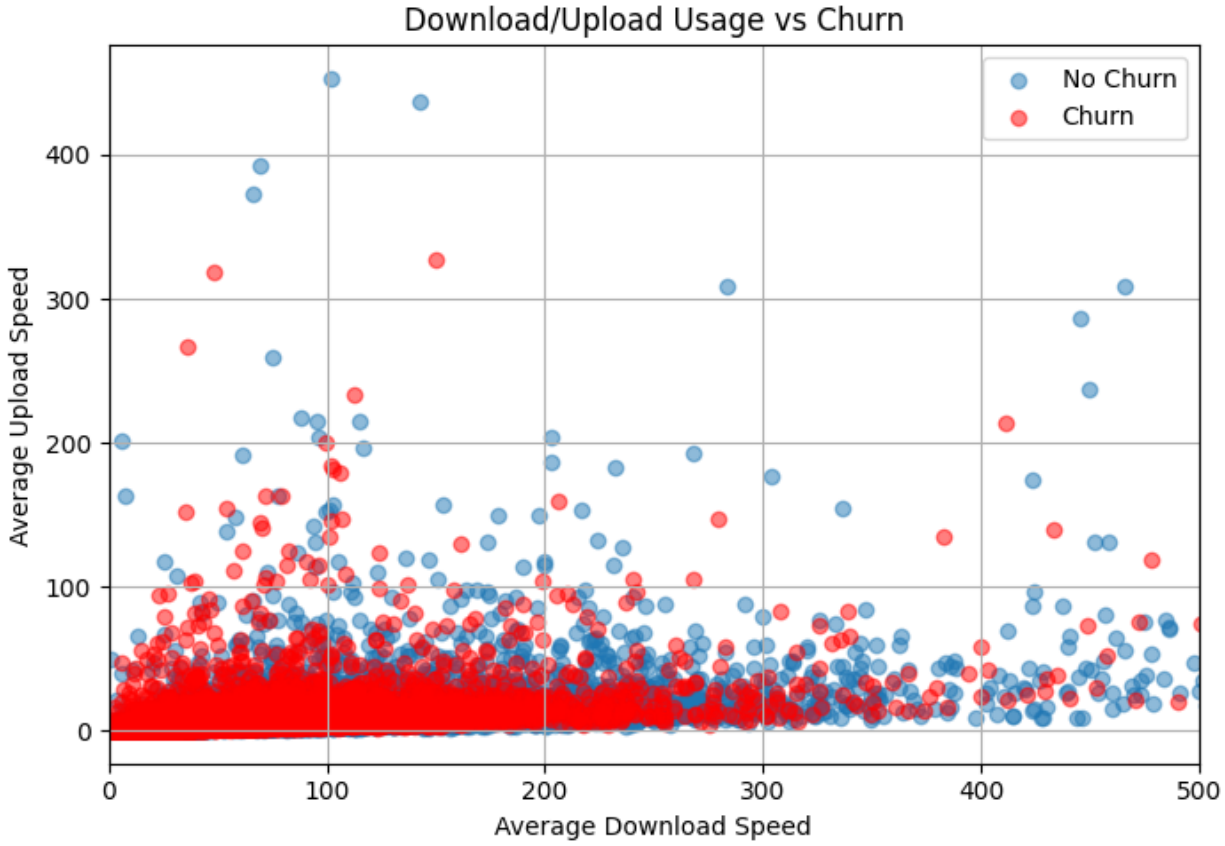
This is an interesting graph because we actually have more customers churning when they are not paying as much in their monthly bill averages. We would expect the opposite, a customer thinks they are paying too much for what they are receiving. However, we see a trend that the lower paying customers tend to churn more, perhaps indicating they are looking for deals in the industry and perhaps found one, hence leaving this ISP.

**EDA 2: Churn Distribution**




This piechart shows how the churn numbers are distributed. This is important for machine learning because it can effect our accuracy levels if there are a lot of churners or not a lot of churners. We see this data is pretty even with a bit of skew towards not churning. This gives us a better idea of churn behavior and a better picture of what to expect when running our models.

**EDA 3: Usage and Churn Scatterplot**




This shows a trend of churn customers being clustered under 25 upload speed, over 25 upload speed we see a surplus of satisfied customers. Another trend is that a lot of churn happens below 300 download speed, over 300 we see more satisfaction. Perhaps people who do not utilize their speeds churn more.

# Modeling:

We will discuss how we can answer our questions and use machine learning models to answer these questions and provide solutions at the end.

## Max's Section

The first thing we wanted to determine from the dataset was if we could predict the estimated total value of a customer's current contract (how much they would pay over the entirety of their current contract combined with how much they have already paid over their history with the company) based on other variables. We created this new column 'clv' based on the 'bill_avg', 'remaining_contract', and 'subscription_age' columns.

### 1. Create the new column '*clv*' (customer lifetime value)

```
from pyspark.sql.functions import when, col


isp_df = isp_df.withColumn(
'clv',
when(
col('churn') == 1,
col('bill_avg') * 12 * col('subscription_age')
).when(
col('remaining_contract') == -1,
col('bill_avg') * 12 * col('subscription_age')
).otherwise(
col('bill_avg') * 12 * (col('subscription_age') + col('remaining_contract'))
)
)
display(isp_df)
```

### 2. Import required libraries

```
from pyspark.ml.regression import LinearRegression, RandomForestRegressor
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.sql.functions import avg, abs
```

**3. Build models to create predictions**: We created a Linear Regression model as well as a Random Forest model with an 80-20 random split of training and testing data.

```
# Assemble features
```

```
featureAssembler = VectorAssembler(
inputCols=['remaining_contract',
'service_failure_count',
'download_avg',
'upload_avg',
'download_over_limit',
'is_tv_subscriber',
'is_movie_package_subscriber'],
outputCol="features")

model = featureAssembler.transform(isp_df).select("features", "clv")

# Ensure the "features" column is present
model.show()

# Split data
train_data, test_data = model.randomSplit([0.8, 0.2])

# Linear Regression
lr = LinearRegression(featuresCol="features", labelCol="clv")
lr_model = lr.fit(train_data)
lr_preds = lr_model.transform(test_data)

# Random Forest
rf = RandomForestRegressor(featuresCol="features", labelCol="clv", seed=42)
rf_model = rf.fit(train_data)
rf_preds = rf_model.transform(test_data)
```

**4. Get metrics of performance to test models**: We used root mean squared error (RMSE), mean absolute error (MAE), and $R^2$ to test the accuracy of our model.

```
# Root Mean Squared Error (RMSE)
rmse = RegressionEvaluator(labelCol="clv", predictionCol="prediction", metricName="rmse")
rmse_lr = rmse.evaluate(lr_preds)

# Mean Absolute Error (MAE)
mae = RegressionEvaluator(labelCol="clv", predictionCol="prediction", metricName="mae")
mae_lr = mae.evaluate(lr_preds)

# Evaluate
r2_lr = RegressionEvaluator(labelCol="clv", predictionCol="prediction", metricName="r2").evaluate(lr_preds)

# Evaluate
r2_rf = RegressionEvaluator(labelCol="clv", predictionCol="prediction", metricName="r2").evaluate(rf_preds)
```

```
# Root Mean Squared Error (RMSE)
rmse_rf = rmse.evaluate(rf_preds)

# Mean Absolute Error (MAE)
mae_rf = mae.evaluate(rf_preds)
```

Output:

Linear Regression Performance:

R-squared: 0.1507

RMSE: 691.60

MAE: 443.87

Random Forest Performance:

R-squared: 0.1693

RMSE: 683.99

MAE: 431.85

Based on the performance scores, we were able to conclude the models were performing very poorly, with a very low R-squared percentage despite inclusion of potential data leakage in the model, as well as very high levels of RMSE and MAE.

**5. Modify data to improve scores**: To try and improve these scores, we decided to modify the data by excluding instances where clv is 0, as well as adding the column 'contract_status_index' to the model to include more potential impactors.

```
# Remove 0s from clv
isp_df = isp_df.filter(isp_df['clv'] != 0)
```

**6. Rebuild models to create predictions**: We rebuilt our Linear Regression model and Random Forest model with these changes to see if the models were predicting clv better.

```
# Assemble features
featureAssembler = VectorAssembler(
inputCols=['remaining_contract',
'service_failure_count',
'download_avg',
'upload_avg',
'download_over_limit',
'is_tv_subscriber',
'is_movie_package_subscriber',
# Add contract_status_index
'contract_status_index'],
outputCol="features")

model = featureAssembler.transform(isp_df).select("features", "clv")

# Ensure the "features" column is present
model.show()

# Split data
train_data, test_data = model.randomSplit([0.8, 0.2])

# Linear Regression
lr = LinearRegression(featuresCol="features", labelCol="clv")
lr_model = lr.fit(train_data)
lr_preds = lr_model.transform(test_data)

# Random Forest
rf = RandomForestRegressor(featuresCol="features", labelCol="clv", seed=42)
rf_model = rf.fit(train_data)
rf_preds = rf_model.transform(test_data)
```

**7. Get new metrics to test performance**: Here, we re-tested the models' accuracies with the same evaluators.

```
# Root Mean Squared Error (RMSE)
rmse = RegressionEvaluator(labelCol="clv", predictionCol="prediction", metricName="rmse")
rmse_lr = rmse.evaluate(lr_preds)

# Mean Absolute Error (MAE)
mae = RegressionEvaluator(labelCol="clv", predictionCol="prediction", metricName="mae")
mae_lr = mae.evaluate(lr_preds)
```

```
# Evaluate
r2_lr = RegressionEvaluator(labelCol="clv", predictionCol="prediction", metricName="r2").evaluate(lr_preds)

# Evaluate
r2_rf = RegressionEvaluator(labelCol="clv", predictionCol="prediction", metricName="r2").evaluate(rf_preds)

# Root Mean Squared Error (RMSE)
rmse_rf = rmse.evaluate(rf_preds)

# Mean Absolute Error (MAE)
mae_rf = mae.evaluate(rf_preds)
```

Output:

Linear Regression Performance:

R-squared: 0.1350

RMSE: 630.16

MAE: 428.21

Random Forest Performance:

R-squared: 0.1607

RMSE: 620.72

MAE: 422.44

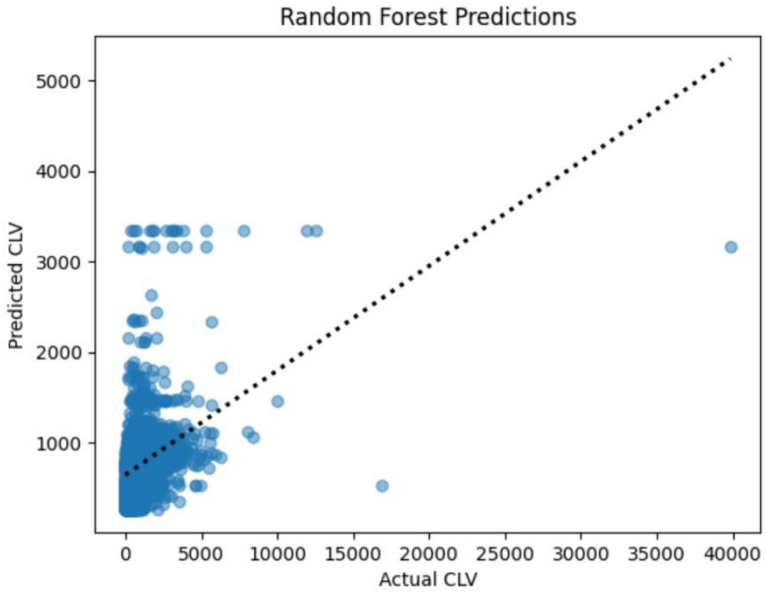
Despite the narrowing down of the data, both models did not improve in their predictive abilities to find clv based on the given features. The models were still very poor predictors.

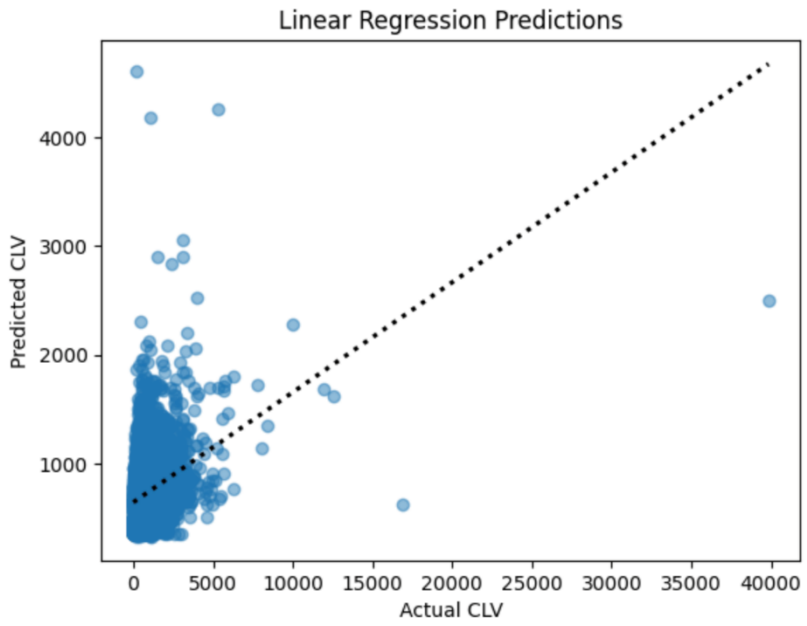
**8. Linear Regression Predictions Visualization**: Despite the models being poor predictors according to our test features, we decided to make correlation graphs that showed model prediction versus actual outcome, with a best fit line displaying perfect performance.

```
import seaborn as sns
import matplotlib.pyplot as plt
```

```
lr_df = lr_preds.filter(lr_preds.clv <= 15000).toPandas()

sns.regplot(data=lr_df, x='clv', y='prediction', line_kws={"color": "black", "linestyle": "dotted"},
scatter_kws={"alpha":0.5}, ci=None)
plt.title('Linear Regression Predictions')
plt.xlabel('Actual CLV')
plt.ylabel('Predicted CLV')
plt.show()

rf_df = rf_preds.filter(rf_preds.clv <= 15000).toPandas()
sns.regplot(data=rf_df, x='clv', y='prediction', line_kws={"color": "black", "linestyle": "dotted"},
scatter_kws={"alpha":0.5}, ci=None)
plt.title('Random Forest Predictions')
plt.xlabel('Actual CLV')
plt.ylabel('Predicted CLV')
plt.show()
```

**Output:**

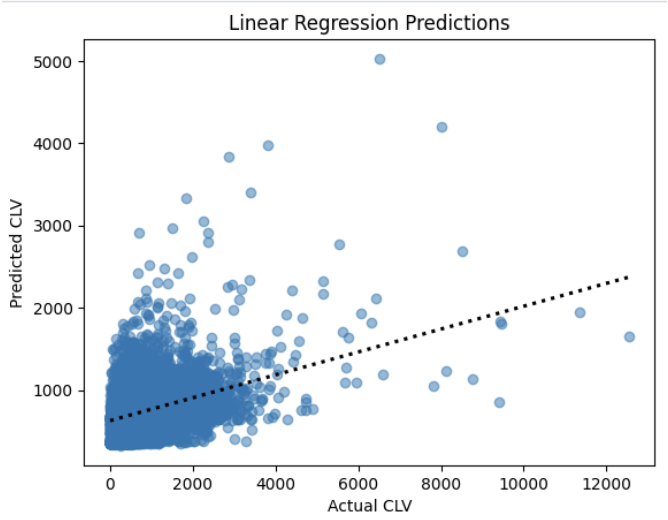


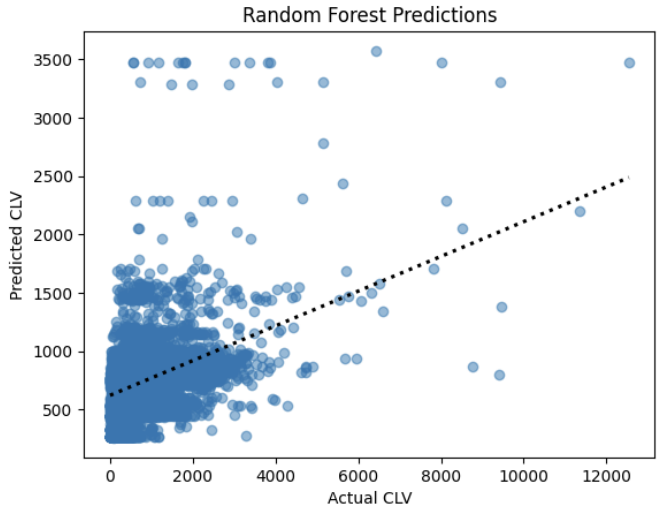



The graphs illustrated similar results, however we were curious if removing outliers would improve graph prediction, so we recreated the graphs.

```
import seaborn as sns
import matplotlib.pyplot as plt


lr_df = lr_preds.filter(lr_preds.clv <= 15000).toPandas()


sns.regplot(data=lr_df, x='clv', y='prediction', line_kws={"color": "black", "linestyle": "dotted"},
scatter_kws={"alpha":0.5}, ci=None)
plt.title('Linear Regression Predictions')
plt.xlabel('Actual CLV')
plt.ylabel('Predicted CLV')
plt.show()


rf_df = rf_preds.filter(rf_preds.clv <= 15000).toPandas()
sns.regplot(data=rf_df, x='clv', y='prediction', line_kws={"color": "black", "linestyle": "dotted"},
scatter_kws={"alpha":0.5}, ci=None)
plt.title('Random Forest Predictions')
plt.xlabel('Actual CLV')
plt.ylabel('Predicted CLV')
plt.show()
```

**Output:**




This unfortunately did not improve our findings or create a better illustrated graph.

## **Conclusion**:

Based on our attempts to predict customer lifetime value, we believe that the data does not include enough strong predictors to create a model to accurately predict CLV. This means that the variables used, such as download and upload average and service failure count do not have a strong impact on the total value of a customer based on their current and past contracts. With more impactful monetary data, this model would likely be much more predictive and accurate. However, with the data provided from the dataset, predicting the estimated total value of a customer's current contract would be very difficult and inaccurate.

**Continue to Next Model**

# Subscription Age and Internet Usage Effect on Churn

**Introduction:**

This section investigates how subscription age and internet usage influence whether customers are likely to churn. By building and evaluating predictive models using PySpark, I aim to quantify the relationship between these variables and churn outcomes. I begin with a logistic regression model to establish a baseline understanding of these features' predictive power. I then introduce a more flexible Random Forest model to enhance performance, assess feature importance, and use cross-validation to fine-tune the model for best accuracy. Through this analysis, we gain insight into which behavioral factors are more strongly associated with customer retention, ultimately supporting data-driven strategies for reducing churn.

**Logistic Regression Modeling:**

**Purpose:** Builds a logistic regression model using PySpark to analyze how internet usage and subscription age impact customer churn.

1) **Import Required Libraries: t**hese imports enable feature transformation, classification, model evaluation, and basic column operations using Spark.

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.sql.functions import col
```

2) **Create Combined Feature "internet_usage":** adds a new column internet_usage by summing upload and download averages to quantify total internet consumption.

```
isp_df = isp_df.withColumn("internet_usage", col("upload_avg") + col("download_avg"))
```

3) **Select Modeling Features:** filters the Data Frame to include only necessary columns and removes any rows with missing values.

```
model_df = isp_df.select("subscription_age", "internet_usage", "churn").dropna()
```

4) **Assemble Features into a Single Vector:** combines both input features into a single vector required for Spark ML models

```
assembler = VectorAssembler(
inputCols=["subscription_age", "internet_usage"],
outputCol="features"
)
```

5) **Initialize Logistic Regression Model:** sets up a logistic regression model for binary classification (churn vs. no churn).

```
lr = LogisticRegression(
labelCol="churn",
featuresCol="features",
rawPredictionCol="rawPrediction",
family="binomial"
)
```

6) **Create and Fit Pipeline: w**raps the feature assembler and model into a pipeline and fits it to the dataset.

```
pipeline = Pipeline(stages=[assembler, lr])


model = pipeline.fit(model_df)
```

7) **Output Model Coefficients and Intercept:** extracts and prints the learned coefficients and intercept from the logistic regression model.

```
lr_model = model.stages[-1]
print("coefficients:", lr_model.coefficients)
print("intercept:", lr_model.intercept)
```

8) **Evaluate Model with AUC (Area Under Curve):** Applies the model to the data and evaluates performance using AUC, which measures model discrimination ability between churned and non-churned users.

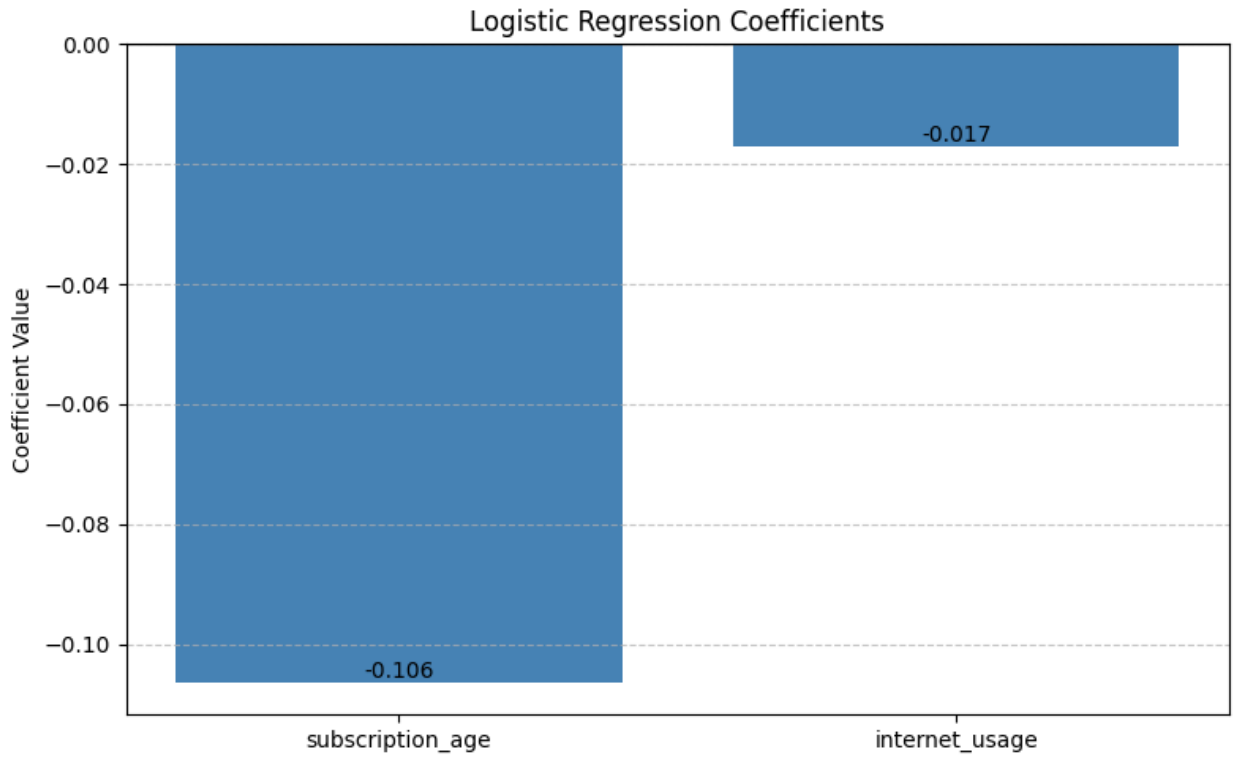
```
predictions = model.transform(model_df)
evaluator = BinaryClassificationEvaluator(labelCol="churn", rawPredictionCol="rawPrediction")
auc = evaluator.evaluate(predictions)
print(f"auc: {auc:.4f}")
```

**9) Create a bar graph to compare coefficients:** this code extracts the logistic regression coefficients for subscription_age and internet_usage and stores them in a DataFrame. It then creates a bar chart to visually display the magnitude and direction (positive or negative) of each feature's influence on churn.

```
import matplotlib.pyplot as plt
```

```
import pandas as pd

# coefficients and intercept
features = ["subscription_age", "internet_usage"]
coefficients = lr_model.coefficients.toArray()
intercept = lr_model.intercept

# create DataFrame for easier handling
coef_df = pd.DataFrame({
"Feature": features,
"Coefficient": coefficients
})

# Bar Chart of Coefficients
plt.figure(figsize=(8, 5))
bars = plt.bar(coef_df["Feature"], coef_df["Coefficient"], color="steelblue")
plt.axhline(0, color='gray', linewidth=1)
plt.title("Logistic Regression Coefficients")
plt.ylabel("Coefficient Value")
plt.grid(axis='y', linestyle='--', alpha=0.7)

# annotate bars
for bar in bars:
yval = bar.get_height()
plt.text(bar.get_x() + bar.get_width()/2, yval, f'{yval:.3f}', ha='center', va='bottom')

plt.tight_layout()
plt.show()
```

**Output:**




**Logistic model conclusions:** both features are negatively associated with churn. Longer-tenured customers and high internet users are less likely to churn. Subscription age having a larger impact in this model. The model is fairly predictive (AUC 0.768).

**Random Forest Model:**

**Purpose:** Introduces a second machine learning model to compare results with logistic regression.

1) **Import Necessary Libraries:**

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml import Pipeline
from pyspark.ml.evaluation import BinaryClassificationEvaluator
```

2) **Assemble Features:** This code combines the subscription_age and internet_usage columns into a single feature vector.

```
assembler = VectorAssembler(
inputCols=["subscription_age", "internet_usage"],
```

```
outputCol="features"
)
```

3) **Train-Test Split:** splits the dataset into 80% training and 20% testing.

```
train_df, test_df = model_df.randomSplit([0.8, 0.2], seed=42)
```

4) **Define Random Forest Model:** This code defines a Random Forest classifier in PySpark to predict customer churn. It uses 100 decision trees with a maximum depth of 5, taking the features column as input and churn as the target label.

```
rf = RandomForestClassifier(
labelCol="churn",
featuresCol="features",
numTrees=100,
maxDepth=5,
seed=42
)
```

5) **Create and Train Pipeline:** creates a machine learning pipeline that first assembles the features and then applies the Random Forest model. It then trains the pipeline on the training dataset to produce a fitted model.

```
pipeline = Pipeline(stages=[assembler, rf])
model = pipeline.fit(train_df)
```

6) **Generate Predictions and Evaluate AUC:** applies the model to test data and evaluates it using AUC.

```
predictions = model.transform(test_df)

evaluator = BinaryClassificationEvaluator(
labelCol="churn",
rawPredictionCol="rawPrediction",
metricName="areaUnderROC"
)
auc = evaluator.evaluate(predictions)

print(f"random forest auc: {auc:.4f}")
```

**Random Forest model conclusion:** The Random Forest model achieved a significantly higher AUC of 0.7961 compared to the logistic regression model. Next, I plan to visualize

feature importance and further optimize the model's performance through cross-validation.

**Feature Importance in Random Forest Model:**

**Purpose:** Introduces the next analysis step. Which is interpreting feature importance scores from the trained Random Forest model.

1) **Extract the Trained Model:**

```
rf_model = model.stages[-1]
```

2) **Get Feature Importances:** retrieves the importance scores assigned to each input feature by the trained Random Forest model. The scores are converted into an array to make them easier to interpret and display.

```
importances = rf_model.featureImportances.toArray()
```

3) **Pair with Feature Names and Print:** pairs each feature name with its corresponding importance score from the model. It then prints out how much each feature contributed to predicting churn.

```
feature_names = ["subscription_age", "internet_usage"]
feature_importance_dict = dict(zip(feature_names, importances))


for feature, importance in feature_importance_dict.items():
print(f"{feature}: {importance:.4f}")
```

**Feature importance conclusion:** The feature importance analysis showed that internet usage (0.8983) plays a far greater role in predicting churn compared to subscription age (0.1017), showing that recent usage behavior is a much stronger indicator of customer retention risk than how long a customer has been subscribed.

**Cross Validation**

**Purpose:** Introduces the model tuning stage using cross-validation to optimize Random Forest performance by testing different parameter combinations.

1) **Import Required Libraries:**

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml import Pipeline
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from pyspark.ml.evaluation import BinaryClassificationEvaluator
```

2) **Prepare Feature Vector and Data Split:** assembles the input features into a single vector required for modeling. It then splits the dataset into 80% training and 20% testing sets using a fixed random seed for reproducibility.

```
assembler = VectorAssembler(
inputCols=["subscription_age", "internet_usage"],
outputCol="features"
)

train_df, test_df = model_df.randomSplit([0.8, 0.2], seed=42)
```

3) **Defines Random Forest Classifier:** starts a Random Forest classifier that will use the features column to predict the churn label. The seed ensures consistent results across runs.

```
rf = RandomForestClassifier(
labelCol="churn",
featuresCol="features",
seed=42
)
```

4) **Create ML Pipeline:** creates a machine learning pipeline that first assembles the features and then applies the Random Forest classifier.

```
pipeline = Pipeline(stages=[assembler, rf])
```

5) **Create Hyperparameter grid:** defines a hyperparameter grid for tuning the Random Forest model. It specifies different values for the number of trees, tree depth, and minimum instances per node to test during cross-validation.

```
paramGrid = ParamGridBuilder() \
.addGrid(rf.numTrees, [50, 100]) \
.addGrid(rf.maxDepth, [3, 5, 7]) \
.addGrid(rf.minInstancesPerNode, [1, 5]) \
.build()
```

6) **Defines Evaluator and Cross Validator:** sets up an evaluator to measure model performance using AUC for binary classification. It also defines a cross-validator to test different hyperparameter combinations and select the best model.

```
evaluator = BinaryClassificationEvaluator(
labelCol="churn",
rawPredictionCol="rawPrediction",
metricName="areaUnderROC"
)

cv = CrossValidator(
estimator=pipeline,
estimatorParamMaps=paramGrid,
evaluator=evaluator,
numFolds=3,
parallelism=2
)
```

7) **Fits Cross-Validated Model and Evaluates AUC:** fits the cross-validated Random Forest model on the training data and selects the best-performing version. It then makes predictions on the test set and evaluates the model's accuracy using AUC.

```
cv_model = cv.fit(train_df)


best_model = cv_model.bestModel
predictions = best_model.transform(test_df)
auc = evaluator.evaluate(predictions)
print(f"best model auc on test data: {auc:.4f}")
```

8) **Displays the optimal parameters selected through cross-validation** pulls the final Random Forest model from the best pipeline and prints the optimal hyperparameters selected during cross-validation. It shows the best values for number of trees, maximum depth, and minimum instances per node.

```
rf_model = best_model.stages[-1]
print("best numtrees:", rf_model.getNumTrees)
print("best maxdepth:", rf_model.getOrDefault("maxDepth"))
print("best mininstancespernode:", rf_model.getOrDefault("minInstancesPerNode"))
```

**Cross validation conclusion:** Using cross-validation, I determined that the best hyperparameters for maximizing AUC in the Random Forest model are: numTrees = 100, maxDepth = 7, and minInstancesPerNode = 5. The final AUC that this optimized random forest model provides is 0.8061.

**Visualizations:**

**Purpose:** this cell creates two visualizations

A) A ROC curve to evaluate the model's classification performance.
B) A feature importance bar chart to interpret which features contribute most to the random forest model.

**1) Import matplotblib**

```
import matplotlib.pyplot as plt
```

2) **Convert Predictions to Pandas and Extract Probabilities:** transforms the Spark Data Frame into a Pandas Data Frame to enable plotting. Extracts the probability of class 1 (churn = 1) from the prediction vector.

```
predictions_pd = predictions.select("churn", "probability").toPandas()
predictions_pd["probability"] = predictions_pd["probability"].apply(lambda x: x[1])
```

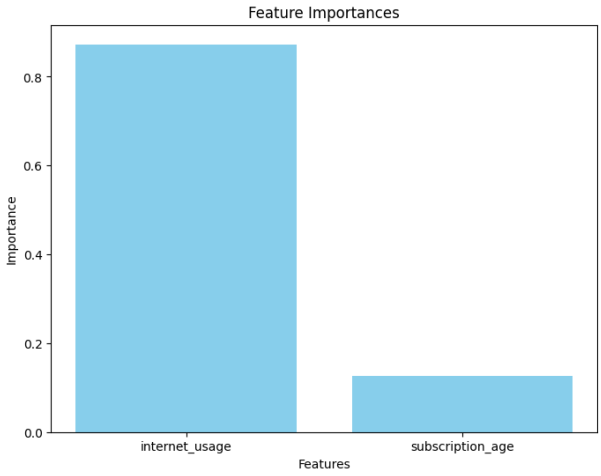
3) **Plot the ROC Curve:** calculates false positive and true positive rates to plot the ROC curve. AUC is displayed as a summary measure of model performance.

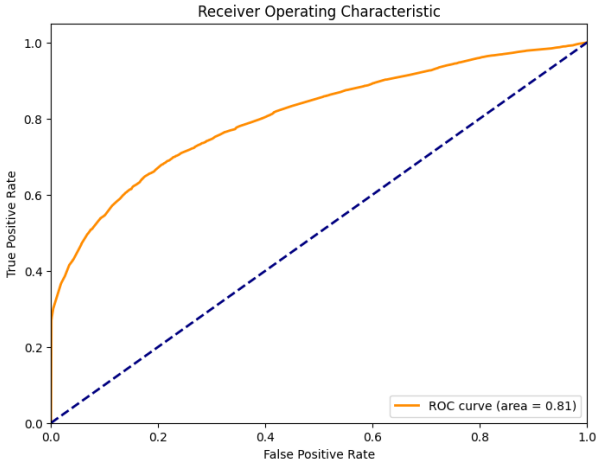
```
from sklearn.metrics import roc_curve, auc


fpr, tpr, _ = roc_curve(predictions_pd["churn"], predictions_pd["probability"])
roc_auc = auc(fpr, tpr)


plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```

4) **Plot Feature Importances from Random Forest Model:** retrieves feature importances from the trained Random Forest model. Matches them to their feature names and creates a Pandas Data Frame. Plots a bar chart to clearly show which feature had greater impact on predicting churn.

```
importances = rf_model.featureImportances
features = ["subscription_age", "internet_usage"]
importances_pd = pd.DataFrame(list(zip(features, importances)), columns=["Feature",
"Importance"]).sort_values(by="Importance", ascending=False)


plt.figure(figsize=(8, 6))
plt.bar(importances_pd["Feature"], importances_pd["Importance"], color='skyblue')
plt.xlabel('Features')
plt.ylabel('Importance')
plt.title('Feature Importances')
plt.show()
```

**Output:**




**Plot Histograms for Visual EDA**

**Purpose:** Generates histograms to visually explore how internet usage and subscription age distributions differ between churned and non-churned customers.

- Uses Seaborn's histplot to generate two subplots:
  - One showing how internet usage varies by churn status
  - One showing how subscription age differs by churn status.

- The plots include kernel density estimates (KDE) to smooth the distribution.
- Different colors distinguish churn status (red and green for churned, blue and orange for retained).

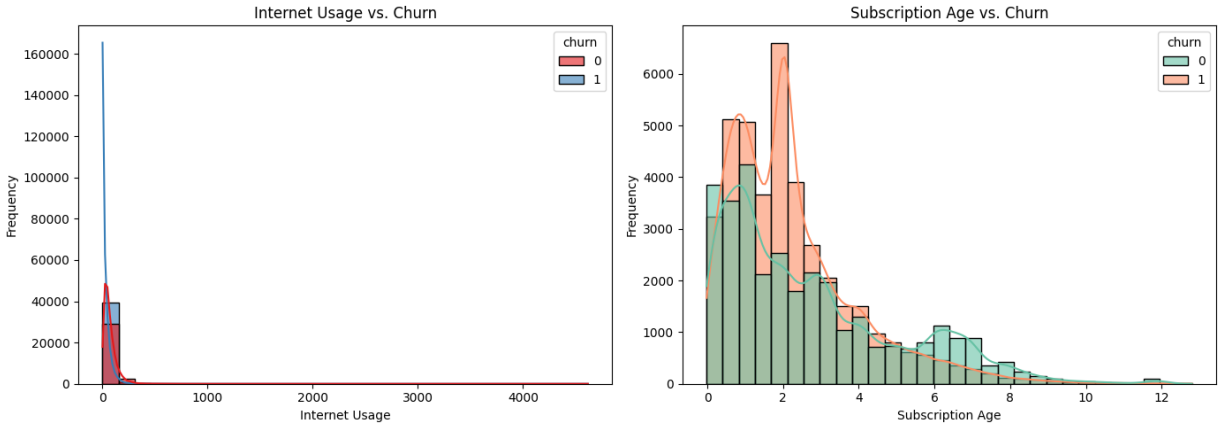
These plots are great for revealing behavioral trends. For example, if churn is more frequent among low-usage or short-tenure customers.

```
import matplotlib.pyplot as plt
import seaborn as sns


plt.figure(figsize=(14, 5))


plt.subplot(1, 2, 1)
sns.histplot(data=plot_df, x="internet_usage", hue="churn", bins=30, kde=True, palette="Set1", alpha=0.6)
plt.title("Internet Usage vs. Churn")
plt.xlabel("Internet Usage")
plt.ylabel("Frequency")

plt.subplot(1, 2, 2)
sns.histplot(data=plot_df, x="subscription_age", hue="churn", bins=30, kde=True, palette="Set2", alpha=0.6)
plt.title("Subscription Age vs. Churn")
plt.xlabel("Subscription Age")
plt.ylabel("Frequency")

plt.tight_layout()
plt.show()
```

**Output:**

0 = remined customers

1= churned customers



**Conclusion:**

Subscription Age: Newer customers are more likely to churn, and Long-tenured users are more stable.

Internet Usage: Very low usage is strongly associated with churn. Moderate-to-high users show better retention

**Continue to Next Model**

# Question 3: Customer Churn Segmentation:

I will be exploring the third question posed in our brainstorming questions phase. This question was: *"How can we segment customer churn based on their usage patterns, subscription-type and behavior?"* I will explore two ways I dealt with this problem: Clustering Model (Unsupervised Learning) and Random Forest (Classification Prediction).

# Model 1- Clustering using K-means:

I first wanted to explore clustering to see how our inputs are related to each other in clusters and find commonalities. To accomplish this, I started with creating clustering features, and cluster on usage patterns. This can be seen in *Figure 1.*

**Lines 1-2:** Import the packages necessary.

**Lines 5-8:** Assemble usage features using VectorAssembler, Output Column will be labeled as usage_features.

**Line 10:** Apply the VectorAssembler transformation to the isp_df and store it as usage_df.

**Lines 13-15:**

**13**- Set up model usage_kmeans to group data into 3 clusters using usage_features from earlier.

**14**- Trains the model on usage_df data to find patterns

**15**- Applies the trained model to label each row in usage_df with a cluster in a new column called usage_column.

**Lines 17-20:** Groups the data by each usage cluster label, then calculates the average of churn and other usage metrics for each cluster, then display it in a table.

This table is not relevant and does not look nice, so I will not include it, this is just a data checking step to make sure calculations load into the variable.

```
1  from pyspark.ml.clustering import KMeans
2  from pyspark.ml.feature import VectorAssembler
3
4  # Assemble usage features
5  usage_assembler = VectorAssembler(
6      inputCols=["download_avg", "upload_avg", "download_over_limit",
       "service_failure_count"],
7      outputCol="usage_features"
8  )
9
10 usage_df = usage_assembler.transform(isp_df)
11
12 # KMeans on usage
13 usage_kmeans = KMeans(featuresCol="usage_features",
   predictionCol="usage_cluster", k=3)
14 usage_model = usage_kmeans.fit(usage_df)
15 usage_clustered = usage_model.transform(usage_df)
16
17 # Check average churn and features by cluster
18 usage_clustered.groupBy("usage_cluster").agg(
19     {"churn": "avg", "download_avg": "avg", "upload_avg": "avg",
       "download_over_limit": "avg", "service_failure_count": "avg"}
20 ).show()
```

**Figure 1:** This code assemblers features trains the data and applies it to each row and assigns clusters in a new column

### *2:* Cluster by Subscription Behavior:

**Lines 1-3:** Combines subscription-related info into one column called scription_features.

**Line 6:** Adds that new subscription_features column to the original data (isp_df) saving it as subscription_df.

**Line 8:** Sets up a model that groups customers into 3 clusters based on subscription behavior.

**Line 9:** Trains the model to find patterns in the subscription data.

**Line 10:** Adds a column to show which subscription cluster each person belongs to

**Lines 12-14:** Shows the average churn and subscription behaviors for each cluster.

```
1  subscription_assembler = VectorAssembler(
2      inputCols=["is_tv_subscriber", "is_movie_package_subscriber",
       "subscription_age"],
3      outputCol="subscription_features"
4  )
5
6  subscription_df = subscription_assembler.transform(isp_df)
7
8  subscription_kmeans = KMeans(featuresCol="subscription_features",
   predictionCol="subscription_cluster", k=3)
9  subscription_model = subscription_kmeans.fit(subscription_df)
10 subscription_clustered = subscription_model.transform(subscription_df)
11
12 subscription_clustered.groupBy("subscription_cluster").agg(
13     {"churn": "avg", "is_tv_subscriber": "avg", "is_movie_package_subscriber":
       "avg", "subscription_age": "avg"}
14 ).show()
15
```

**Figure 2:** This code block assembles features, trains the data and clusters them based on subscription behavior

Again, .show( ) is not a neat viewable table, this is to ensure data is loaded properly.

### *Step 3:* Cluster on Billing Behavior:

**Lines 1-4:** Combines subscription-related info into one column called billing_features.

**Line 6:** Adds that new billing_features column to the original data (isp_df) saving it as billing_df.

**Line 8:** Sets up a model that groups customers into 3 billing-based clusters.

**Line 9:** Trains the model using billing data to find spending patterns.

**Line 10:** Labels each customer with a billing_cluster based on the models result.

**Lines 12-14:** Shows the average churn, bill amount, contract time for each billing cluster.

```
1  billing_assembler = VectorAssembler(
2      inputCols=["bill_avg", "remaining_contract"],
3      outputCol="billing_features"
4  )
5
6  billing_df = billing_assembler.transform(isp_df)
7
8  billing_kmeans = KMeans
   (featuresCol="billing_features",
   predictionCol="billing_cluster", k=3)
9  billing_model = billing_kmeans.fit(billing_df)
10 billing_clustered = billing_model.transform
   (billing_df)
11
12 billing_clustered.groupBy("billing_cluster").agg(
13     {"churn": "avg", "bill_avg": "avg",
       "remaining_contract": "avg"}
14 ).show()
15
```

**Figure 3:** This assembles features related to billing and trains the model to find patterns and groups it into a models result.

Again, .show( ) is not a neat viewable table, this is to ensure data is loaded properly.

### *Step 4:* Convert to Pandas and Average

```
1  usage_pd = usage_clustered.select("usage_cluster", "churn").toPandas()
2  subscription_pd = subscription_clustered.select("subscription_cluster",
   "churn").toPandas()
3  billing_pd = billing_clustered.select("billing_cluster", "churn").toPandas()
4
```

**Line 1:** Takes the usage_cluster and churn columns from usage_clustered and converts them to a pandas DataFrame.

**Figure 4**: Converts each metric into a pandas DataFrame.

**Line 2:** Does the same for subscription data; gets subscription_cluster and churn as pandas DataFrame

**Line 3:** Does the same for billing; grabs billing_cluster and churn as a pandas DataFrame

```
1  # Usage churn rate
2  usage_churn = usage_pd.groupby("usage_cluster").mean(numeric_only=True).
   reset_index()
3
4  # Subscription churn rate
5  subscription_churn = subscription_pd.groupby("subscription_cluster").mean
   (numeric_only=True).reset_index()
6
7  # Billing churn rate
8  billing_churn = billing_pd.groupby("billing_cluster").mean(numeric_only=True).
   reset_index()
9
```

**Lines 1-2:** Calculates the average churn group by each usage cluster using .mean.

**Figure 5:** Calculates the means for each cluster to use in visualization.

**Line 5:** Calculates the average churn for each subscription cluster.

**Line 3:** Calculates the average churn for each billing cluster.

## *Step 5:* Make Visualizations:

**Line 3:** Created 3 side-by-side bar charts with shared y-axis and sets the figure size.

**Line 6:** Plots the churn rates by usage cluster in the first chart

**Lines 7-9:** Adds title and axis labels to the first chart.

**Line 12-14:** Plots and labels churn rates by subscription cluster in the second chart.

**Lines 17-19:** Plots and labels churn rate by billing cluster in third chart.

**Lines 21-22:** Adjusts spacing and shows figure.

```python
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, 3, figsize=(18, 5), sharey=True)

# Usage
axs[0].bar(usage_churn["usage_cluster"], usage_churn["churn"], color='skyblue')
axs[0].set_title("Churn Rate by Usage Cluster")
axs[0].set_xlabel("Usage Cluster")
axs[0].set_ylabel("Avg Churn Rate")

# Subscription
axs[1].bar(subscription_churn["subscription_cluster"], subscription_churn["churn"], color='lightgreen')
axs[1].set_title("Churn Rate by Subscription Cluster")
axs[1].set_xlabel("Subscription Cluster")

# Billing
axs[2].bar(billing_churn["billing_cluster"], billing_churn["churn"], color='salmon')
axs[2].set_title("Churn Rate by Billing Cluster")
axs[2].set_xlabel("Billing Cluster")

plt.tight_layout()
plt.show()
```

**Figure 6:** Uses Matplotlib to graph 3 side by side bar charts to find differences between these three variables.

```python
sns.set(style="whitegrid", palette="muted")

fig, axs = plt.subplots(1, 3, figsize=(18, 5))

# Usage Scatter
sns.scatterplot(
    data=usage_pd,
    x="download_avg",
    y="upload_avg",
    hue="usage_cluster",
    style="churn",
    ax=axs[0]
)
axs[0].set_title("Usage Clusters vs. Churn")
axs[0].set_xlabel("Download Avg")
axs[0].set_ylabel("Upload Avg")

# Subscription Scatter
sns.scatterplot(
    data=subscription_pd,
    x="subscription_age",
    y="is_tv_subscriber",
    hue="subscription_cluster",
    style="churn",
    ax=axs[1]
)
axs[1].set_title("Subscription Clusters vs. Churn")
axs[1].set_xlabel("Subscription Age")
axs[1].set_ylabel("TV Subscriber (0/1)")

# Billing Scatter
sns.scatterplot(
    data=billing_pd,
    x="bill_avg",
    y="remaining_contract",
    hue="billing_cluster",
    style="churn",
    ax=axs[2]
)
axs[2].set_title("Billing Clusters vs. Churn")
axs[2].set_xlabel("Bill Avg")
axs[2].set_ylabel("Remaining Contract")

plt.tight_layout()
plt.show()
```

**Figure 7:** Uses Matplotlib to graph 3 side by side scatter charts to find differences between these three variables.

**Line 4:** Sets a clean white grid style and a muted color palette for all seaborn plots.

**Line 6:** Creates a single row of 3 plots side by side, making figure 18 in. wide and 5 in tall

**Lines 10-16:** Plots usage behavior, each point = customer, color = usage_cluster, shaper = churn or not churn, and axis.

**Line 17-19:** Adds titles and labels for first plot.

**Lines 22-29:** Plots subscription behavior: X-axis: how long someone's been subscribed, Y-axis: whether they subscribe to TV (0,1), Color = subscription_cluster, chape =churn.

**Lines 30-32:** Labels and titles second plot.

Lines 35-42: Plots billing behavior: X-axis: average bill amount, Y-axis: how much contract time is left, Color = billing_cluster, shape = churn.

**Lines 43-45:** Titles and labels the third plot.

**Both visualizations shown on the following page with descriptions**

## Churn Rates for Usage, Subscription and Billing and 3 Group Clustering

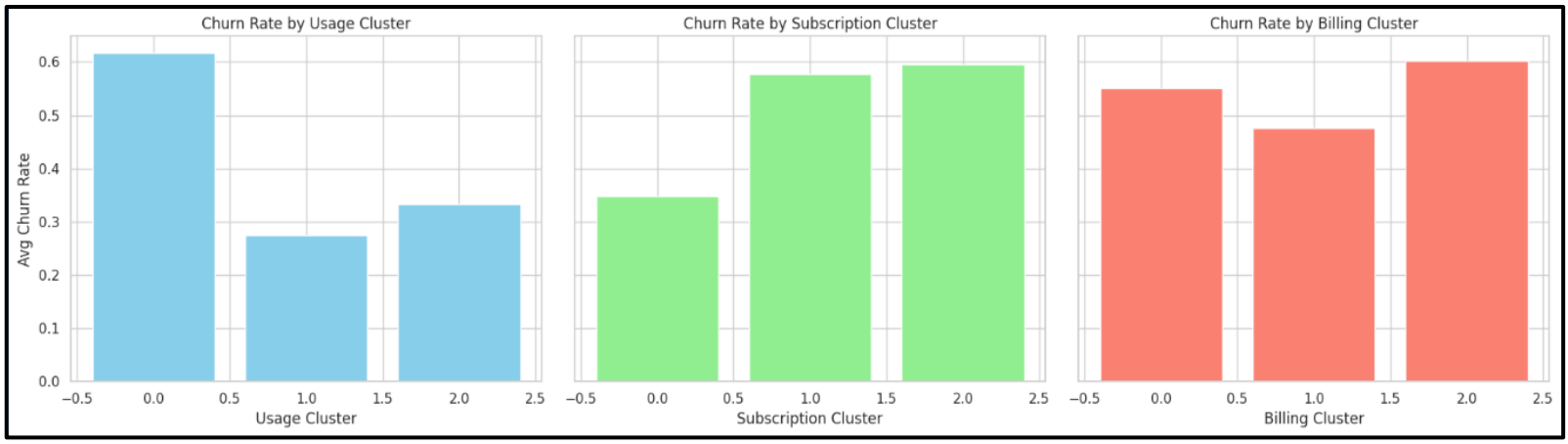



**Figure 8:** Usage, Subscription & Billing variables into 3 clusters based on similar behavior.

**Graph 1- Usage Clustering:** Cluster 0 has the highest churn rate at around 0.62. This cluster is likely to be **heavy users.** May experience more services failure and go over data limits. Takeaway: High-usage customers, high-problem areas can be tied to customers being frustrated and are more likely to leave/churn.

**Graph 2- Subscription:** Clusters 1 & 2 have the higher churns, 0.58-0.6. These clusters could possibly be newer customers or customers that are not subscribed to many services. Cluster 0 had the lowest churn near 0.34. Takeaway: Customers who have been customers longer and use more services tend to stay. New or less engaged users are at higher churn risk.

**Graph 3- Billing:** Cluster 2 has the highest churn near 0.6. This cluster could likely have high bills or may be nearing the end or don't have contracts. Cluster 1 had the lowest churn. Takeaway: High-paying customers near contract end or without contract are most likely to churn, possibly due to re-evaluating cost or service.

## Churn Rates for Usage, Subscription and Billing and 3 Group Clustering

**Group 0 (Light Pink):** This cluster is low usage, upload & download averages, mostly consist of non-churners. This group has low usage and low churn, this cluster is satisfied customers with low churn and minimal risk.

**Group 1 (Medium Pink):** This cluster has moderate churn, slightly higher than cluster 0. This cluster/customer group seems to be a middle ground, with some churn risk as usage starts to increase. This group has the highest number of churn.

**Group 2 (Dark Purple):** This cluster has much higher download and upload ranges, there are more churners in proportion to its cluster showing this is the highest proportion churn.

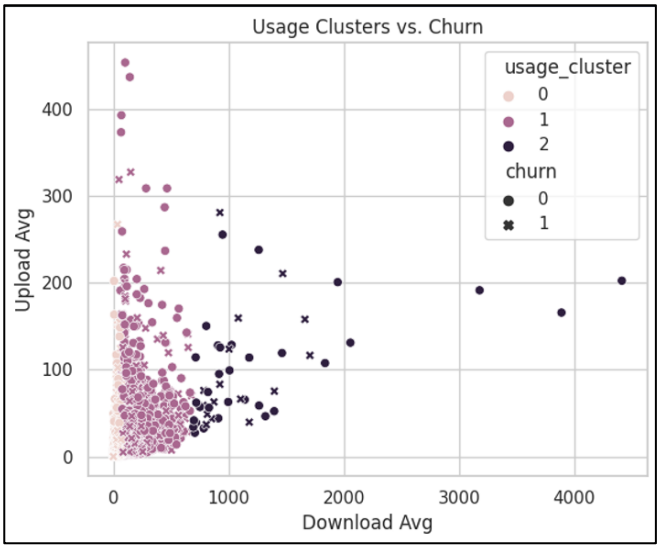



**Figure 9:** Usage clusters on churn rates

**Highest Non-Churners**

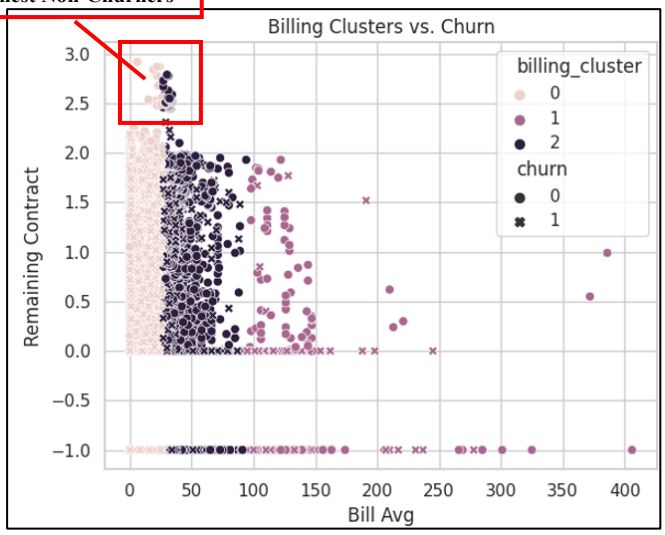



**Figure 10 :** Billing clusters on churn rates

**Group 0 (Light Pink):** This cluster has low bill averages (mostly under $50). This cluster appears to be a lot more customers churning (x's). This might hint at these customers have expiring contracts, and are looking for cheaper or different alternatives.

**Group 1 (Medium Pink):** This cluster is a wider spread for average (up to $400). All customers who had no time in their remaining contracts churned meaning this ISP should focus on offering incentives for this cluster to stay.

**Group 2 (Dark Purple):** This cluster is the moderate bill average (around $30-$100 range). This is a very good mix of churners and non-churners, as we approach end of contracts we see churn more.

**Group 0 (Light Pink):** This cluster are customers who have been with the ISP for a long time- 5+ years. This seems to have the lowest churn rate.

**Group 1 (Medium Pink):** This cluster is a wider spread for age of subscription- 2-5 years. Customers look like they churn more in lower subscription ages, and churn less as the subscription age increase.

**Group 2 (Dark Purple):** This cluster is the lower tier subscription age < 2 years. This group churns the most whether they had a subscription or not. This group is highly likely to churn.

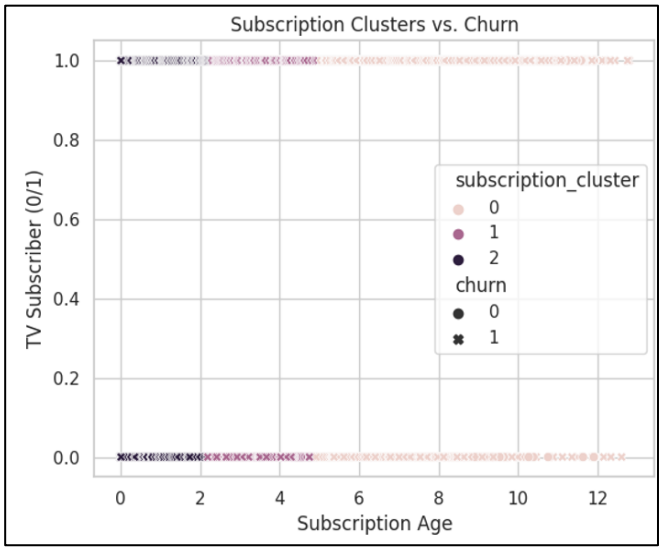



**Figure 11:** Subscription clusters on churn rates

This graph is not the best for these variables. TV subscription does not seem to impact churn neither does age until 10+ years.

# Model 2- Random Forest

**Lines 1-4:** Import necessary packages, rf classifier, binary classifier for churn, vector assembler, validation splits, grid builder.

**Lines 7-21:** Uses VectorAssembler to bundle features into a vector column for the model input. We then list the input features that go into the model and the final output will be 'features' as the output column.

**Line 23:** Applies VectorAssebmler to idp_df, adding a 'features' column that combines the inputs from the previous line of code.

**Line 26:** This randomly splits the data into sets of training (70%) and testing (30%) for model training and evaluation. The seed = 42 ensures reproducibility.

**Line 29**: Creates the random forest classifier object:

featuresCol – vector column created

labelCol – target column for prediction -churn

numTrees=50 – uses 50 decision trees in ensemble

**Line 30:** Trains/fits the random forest model on the training set.

**Line 33:** Applies the trained model to the test set and generates predictions and probability scores.

**Line 36:** Sets up the evaluator to use AUC (Area under the ROC curve) as the evaluation metric.

**Line 37:** Calculates the AUC score on the test predictions—a higher value (closer to 1) indicates a better model performance.

**Line 38:** Prints the AUC score to the output/logs.

**Output**: AUC: 0.9662.

I wanted to make sure that this high AUC isn't a signal of overfitting, so I will now compare the AUC's from the training data and the testing data to ensure that the model is not 'learning' the training data.

```
1  from pyspark.ml.classification import RandomForestClassifier
2  from pyspark.ml.evaluation import BinaryClassificationEvaluator
3  from pyspark.ml.feature import VectorAssembler
4  from pyspark.ml.tuning import TrainValidationSplit, ParamGridBuilder
5
6  # Final VectorAssembler for prediction
7  rf_assembler = VectorAssembler(
8      inputCols=[
9          "is_tv_subscriber",
10         "is_movie_package_subscriber",
11         "subscription_age",
12         "bill_avg",
13         "remaining_contract",
14         "service_failure_count",
15         "download_avg",
16         "upload_avg",
17         "download_over_limit",
18         "contract_status_onehot"
19     ],
20     outputCol="features"
21 )
22
23 isp_df = rf_assembler.transform(isp_df)
24
25 # Train-test split
26 train_df, test_df = isp_df.randomSplit([0.7, 0.3], seed=42)
27
28 # Random Forest
29 rf = RandomForestClassifier(featuresCol="features", labelCol="churn", numTrees=50)
30 rf_model = rf.fit(train_df)
31
32 # Predictions
33 predictions = rf_model.transform(test_df)
34
35 # Evaluation
36 evaluator = BinaryClassificationEvaluator(labelCol="churn", metricName="areaUnderROC")
37 auc = evaluator.evaluate(predictions)
38 print(f"AUC: {auc}")
```

**Figure 12:** Set up and train Random Forest model.

### *Step 2:* Compare Train/Test AUC:

```
1  train_preds = rf_model.transform(train_df)
2  train_auc = evaluator.evaluate(train_preds)
3  print(f"Train AUC: {train_auc}, Test AUC: {auc}")
```

**Figure 13:** Comparing the Train/Test AUC's

**Line 1:** Applies the trained RandomForestClassifier model to the training data (train_df).

**Line 2:** Uses the BinaryClassificationEvaluator to get the AUC score for the training predictions.

**Line 3:** Compare both AUC values and detect if it is overfitting/underfitting.

**Output: Train AUC:**

**Train** AUC: 0.9642

**Test** AUC: 0.96623

These AUC's are fairly similar only around a 0.002 difference. The model performs equally well on both datasets. This indicates the model is generalizing properly and is not overfitting. It seems to pick up on patterns from the data without memorizing noise, and consistently performing on both training and test. I will use Cross-Validation testing to ensure this is not overfitting, just because they are so similar.

### *Step 2.5:* Cross Validation

```
1  from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
2  from pyspark.ml.evaluation import BinaryClassificationEvaluator
3  evaluator = BinaryClassificationEvaluator(labelCol="churn", metricName="areaUnderROC")
4  cv = CrossValidator(
5      estimator=rf,
6      estimatorParamMaps=ParamGridBuilder().build(),
7      evaluator=evaluator,
8      numFolds=5,          # 5-fold cross-validation
9      seed=42
10 )
11 cv_model = cv.fit(train_df)
12 cv_predictions = cv_model.transform(test_df)
13 cv_auc = evaluator.evaluate(cv_predictions)
14 print(f"Cross-Validated Test AUC: {cv_auc}")
```

**Figure 14:** Cross-Validation on Random Forest

**Lines 1-2:** Import the CrossValidation and BCE

**Line 3:** Creates the evaluator, we label churn as the true label column, AUC is going to be the specified metric for model evaluation

**Lines 4-10:** Using CrossValidator estimator is object rf, builds an empty parameter grid, so cross-validation validates across folds. Set the evaluator to the previous evaluator from earlier. 5-fold Cross-Validation.

**Line 11:** Fits cross-validator to training data, picks best AUC across folds

**Line 12:** Applies the best model from CV to the test set to generate predictions

**Line 13:** Calculates AUC of the CV

**Line 14**: Print the AUC score

**Output: Cross-Validation AUC:**

**Testing CV AUC: 0.9662**

**Meaning:** This AUC is consistent with the original test AUC, this means that the model is stable across different folds of training/testing splits, meaning it is not overfitting, it also generalizes well to unseen data. There is no high variance since we confirmed this with the CV.

### *Step 3:* Accuracy

```
1  from pyspark.ml.evaluation import MulticlassClassificationEvaluator
2
3  accuracy_eval = MulticlassClassificationEvaluator(
4      labelCol="churn", predictionCol="prediction", metricName="accuracy"
5  )
6
7  accuracy = accuracy_eval.evaluate(predictions)
8  print(f"Accuracy: {accuracy:.3f}")
```

**Figure 15:** Find accuracy score on RF

**Line 1:** Import the Evaluator

**Line 3:** Creates an evaluator object to compute accuracy:

labelCol=’churn’ - the column with true class labels

predictionCol=’predicition’ - the column with model-predicted labels

metricName=”accuracy” - tells the calculate the accuracy defined as:

**Accuracy** = **Correct Predictions** / **Total Predictions**

**Line 7:** Calculates the accarucy score by comparing churn column to the predicition column in predictions. From the model’s .transform( ) call.

**Line 8:** Prints out the accuracy rounded to 3 decimal places.

**Accuracy Code Output:**

**Accuracy = 0.937**

**Meaning:** This Random Forest model correctly predicted the churn status of 93.7% of the customers in the test data. Out of 100 customers, about 94 predictions were correct, since this data isn’t imbalanced as we saw in the EDA this means this performance is strong.

### Step 4: Confusion Matrix

| Class | Precision | Recall | F1-Score | Support |
|---|---|---|---|---|
| **0 (No-Churn)** | 0.92 | 0.94 | 0.93 | 9,685 |
| **1 (Churn)** | 0.95 | 0.93 | 0.94 | 11,976 |

**Precision:** Of all predicted class X, how many were correct?

Class 1 (churn) precision = 95% of predicted churners were actually churners.

**Recall:** Of all actual class X, how many were correctly predicted?

Class 1 recall = 93% of actual churners were correctly identified.

**F1-Score:** Balance of precision and recall. F1 = 0.94 for both classes shows balanced, strong performance.

This model is highly accurate, well-balanced between precision and recall, does not overfit, and works with class imbalance.

### *Step 5:* Importances

**Base_features:** This assembles the same features into a variable.

**Expanded_features:** I am adding two one-hot encoded contract status features to the list. This will match the full vector used in VectorAssembler.

**Importances**: This extracts the feature importances from the trained Random Forest model, I used .toArray( ) to a numPy array, each importance value corresponds to feature in expanded_features.

Creates a Pandas DataFrame with two columns: feature names and importance scores. It then sort it by importance (highest to lowest)

Then Visualize this using matplotlib.

```
base_features = [
    "is_tv_subscriber",
    "is_movie_package_subscriber",
    "subscription_age",
    "bill_avg",
    "remaining_contract",
    "service_failure_count",
    "download_avg",
    "upload_avg",
    "download_over_limit"
]

# Expand the one-hot encoded vector
expanded_features = base_features + ["contract_status_onehot_0", "contract_status_onehot_1"]



importances = rf_model.featureImportances.toArray()

import pandas as pd

feature_importance_df = pd.DataFrame({
    "feature": expanded_features,
    "importance": importances
}).sort_values(by="importance", ascending=False)

feature_importance_df


import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(10, 5))
sns.barplot(data=feature_importance_df, x="importance", y="feature", palette="viridis")
plt.title("Random Forest Feature Importances")
plt.xlabel("Importance")
plt.ylabel("Feature")
plt.tight_layout()
plt.show()
```

**Figure 16:** Find importances of features for our Random Forest model.

## Importance Graph:

Remaining contract has the highest important, perhaps taking over too much over the others. This makes sense from the EDA phase we showed if someone has a contract they are not currently churning because they still have contract. I plan on removing this variable and re-doing the Random Forest model to see the actual feature importances.

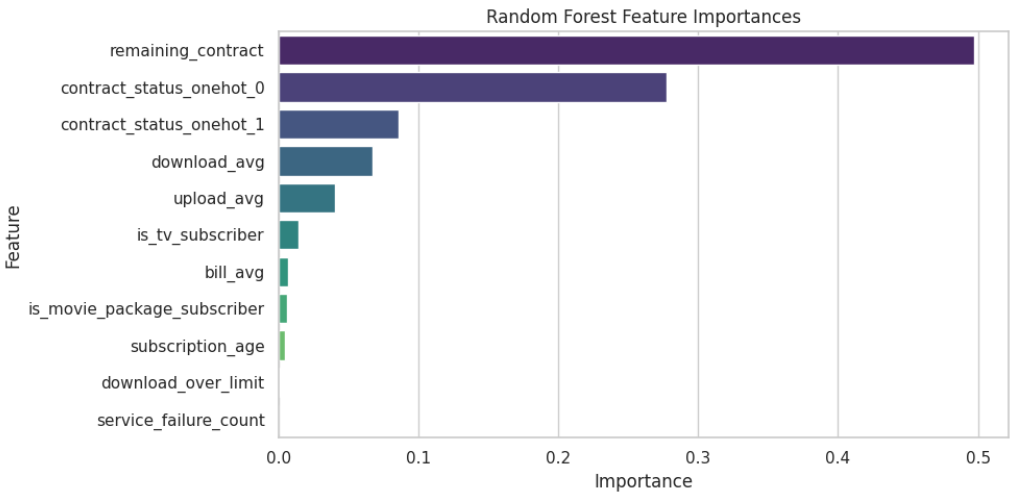



**Figure 17:** Graph the Importances to visualize these importance features.

# Model 3- Random Forest No ‘Remaining Contract’

I will not go line by line, this as already been explained, it is just leaving one feature out.

First, in figure __, we assemble the same features just leaving out ‘remaining_contract’

We then used vector assembler and brought them together as ‘features’

We then split the train and test set, by 70% and 30% and limit the number of trees to 50, with the emphasis on finding ‘churn’

We then fit the model to the new features and make predictions

```
rf_features_no_contract = [
    "is_tv_subscriber",
    "is_movie_package_subscriber",
    "subscription_age",
    "bill_avg",
    "service_failure_count",
    "download_avg",
    "upload_avg",
    "download_over_limit"
]
```

```
from pyspark.ml.feature import VectorAssembler

rf_assembler_no_contract = VectorAssembler(
    inputCols=rf_features_no_contract,
    outputCol="features"
)


isp_df_no_contract = rf_assembler_no_contract.transform(isp_df)
```

```
train_df_nc, test_df_nc = isp_df_no_contract.randomSplit([0.7, 0.3], seed=42)

from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator

rf_nc = RandomForestClassifier(
    featuresCol="features",
    labelCol="churn",
    numTrees=50
)

rf_model_nc = rf_nc.fit(train_df_nc)
predictions_nc = rf_model_nc.transform(test_df_nc)
```

**Figure 18:** Remove Remaining Contract from Model.

***Step 2:*** **Find AUC Score**

We create the evaluator and find the AUC score:

```
evaluator = BinaryClassificationEvaluator(labelCol="churn", metricName="areaUnderROC")
auc_nc = evaluator.evaluate(predictions_nc)

print(f"AUC without remaining_contract: {auc_nc}")
```

**Figure 19:** Finding the AUC score for new model.

**AUC without remaining_contract**: 0.85

This is a lot worse than the other model, which is interesting because it seems like the model relies heavily on these remaining_contract feature. This new model probably performs better without reliance of remaining_contract.

### *Step 3:* Find Accuracy and Error Metrics

We then find the Accuracy of this model:

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

accuracy_eval = MulticlassClassificationEvaluator(
    labelCol="churn", predictionCol="prediction", metricName="accuracy"
)

accuracy = accuracy_eval.evaluate(predictions_nc)
print(f"Accuracy: {accuracy:.3f}")
```

**Figure 20:** Finding accuracy of this new model.

**Accuracy: 0.771**

This is lower than the model with remaining_contract, meaning remaining_contract was a highly predictive feature. This makes sense and may limit this models accuracy and might not be best to find why a customer specifically churns.

We then create predictions by churn and using predictions, to find the error metrics.

```
predictions_nc.groupBy("churn", "prediction").count().show()
```

(2) Spark Jobs

```
+-----+----------+-----+
|churn|prediction|count|
+-----+----------+-----+
|    1|       0.0| 3189|
|    0|       0.0| 7917|
|    1|       1.0| 8787|
|    0|       1.0| 1768|
+-----+----------+-----+
```

**Figure 21:** Finding confusion matrix of new model.

```
from sklearn.metrics import classification_report

# Convert to pandas
pdf_rf = predictions_nc.select("churn", "prediction").toPandas()

# Generate classification report
print(classification_report(pdf_rf["churn"], pdf_rf["prediction"]))
```

**This gives us the confusion matrix for churners and non-churners, the report is shown below:**

| Churn? | Precision | Recall | F1-Score | Support |
|---|---|---|---|---|
| **0** | 0.71 | 0.82 | 0.76 | 9685 |
| **1** | 0.83 | 0.73 | 0.78 | 11976 |

**Class 0 (Non-Churners):**

**Precision:** 71% -- 71% of predicted non-churners was correctly picked by the model.

**Recall:** 82% -- The model correctly identified 82% of all actual non-churners.

**F1-score:** 0.76 – Good balance of precision and recall.

**Class 1 (Churners):**

**Precision:** 83% -- 83% of predicted churners were correct.

**Recall:** 73% -- The model correctly picked 76% of all actual churners.

**F1-Score:** 0.78 – Strong performance overall.

**Conclusion:**

I would choose the model with remaining_contract if my goal is to maximize predictive performance, capture churners early for retention and minimize false churn predictions.

I would chose the model without remaining_contract if I want to look at a scenario without contract data- for customers not in a contract, or testing without a dominant feature.

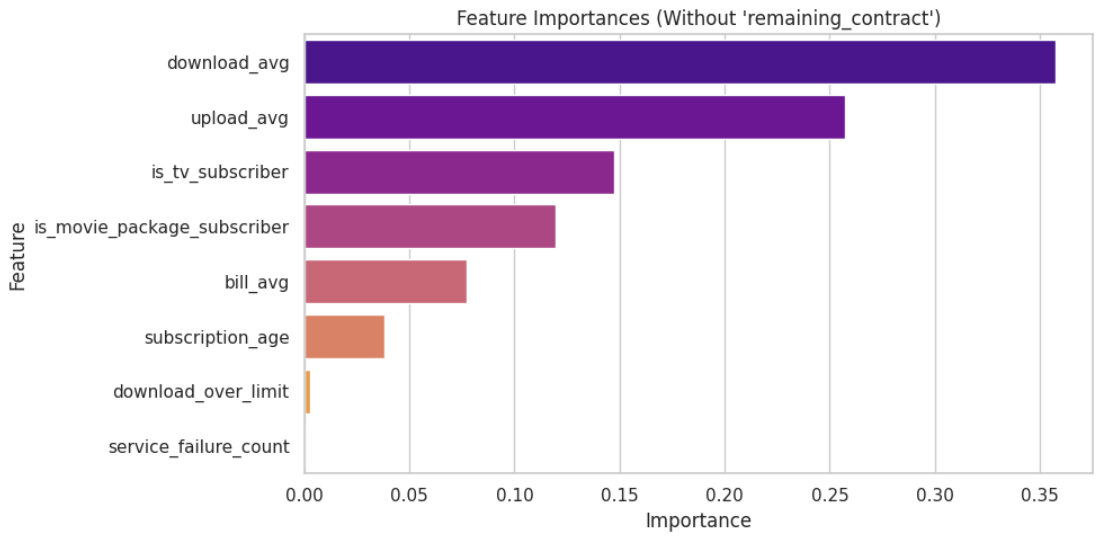



**Figure 22:** Finding confusion matrix of new model.

# Model 4- Logistic Regression:

We just assembled the features columns to use in the Logistic Regression model.

Import the VectorAssembler and assemble the features outputting it to 'lr_features'.

Apply the vector assembler transformation to the isp_dfmand stores it as as isp_df_lr.

Then we create the Logistic Regression, to predict churn, and fit it to the new isp_df_lr.

Split the data into train and test (70% to 30%) and make the predictions using the testing data.

```python
from pyspark.ml.classification import LogisticRegression


feature_cols = [
    "is_tv_subscriber",
    "is_movie_package_subscriber",
    "subscription_age",
    "bill_avg",
    "remaining_contract",
    "service_failure_count",
    "download_avg",
    "upload_avg",
    "download_over_limit",
    "contract_status_onehot"  # use this whole vector as-is
]


from pyspark.ml.feature import VectorAssembler

assembler = VectorAssembler(
    inputCols=feature_cols,
    outputCol="lr_features"
)

isp_df_lr = assembler.transform(isp_df)

# Then update your LogisticRegression to use this
lr = LogisticRegression(featuresCol="lr_features", labelCol="churn")
lr_model = lr.fit(isp_df_lr)

train_lr, test_lr = isp_df_lr.randomSplit([0.7, 0.3], seed=42)
predictions_lr = lr_model.transform(test_lr)
```

**Figure 23**: Created new features for our logistic regression model.

```python
coeffs = lr_model.coefficients.toArray()
intercept = lr_model.intercept
```

**Figure 24**: Build the coefficient variables and intercept variables.

We create model coefficients for each feature and tells us the strength and direction of each features effect on the log-odds of churn.

The intercept will retrieve the intercept term, it will represent the baseline log-odds of churn when all features are zero.

## Make a Feature List to Start Odds Analysis

**Part 1:**

I need to reconstruct the actual feature names after applying one-hot encoding. After one hot encoding we get a single vector column. But when looking at model coefficients or feature importances, I need a full list of flattened feature names.

We first convert the one hot encoded vector into an array to count how many dummy variables were created.

Then we find out how many separate features are inside that one hot encoded vector.

**Part 2:**

Generate readable names for each component of the one hot encoder vector contract_status_onehot (1,0).

Then combine the regular features and the unpacked one-hot features into a full flattened list to get readable feature names.

```
from pyspark.ml.functions import vector_to_array

# Convert onehot vector to array to inspect length
isp_df_temp = isp_df.withColumn("contract_status_onehot_array", vector_to_array("contract_status_onehot"))
onehot_size = len(isp_df_temp.select("contract_status_onehot_array").first()["contract_status_onehot_array"])

(1) Spark Jobs
isp_df_temp: pyspark.sql.dataframe.DataFrame
```

```
Just now (<1s)                                    99

base_features = [
    "is_tv_subscriber",
    "is_movie_package_subscriber",
    "subscription_age",
    "bill_avg",
    "remaining_contract",
    "service_failure_count",
    "download_avg",
    "upload_avg",
    "download_over_limit"
]

# Expand the one-hot vector
expanded_onehot = [f"contract_status_onehot_{i}" for i in range(onehot_size)]

# Final feature names
feature_names = base_features + expanded_onehot
```

```
import pandas as pd
import numpy as np

weights_df = pd.DataFrame({
    "feature": feature_names,
    "coefficient": coeffs
})

# Optional: Add odds ratios
weights_df["odds_ratio"] = weights_df["coefficient"].apply(lambda x: round(np.exp(x), 3))

# Sort by absolute value of coefficients for impact
weights_df = weights_df.sort_values(by="coefficient", key=abs, ascending=False)

print("Intercept:", intercept)
weights_df
```

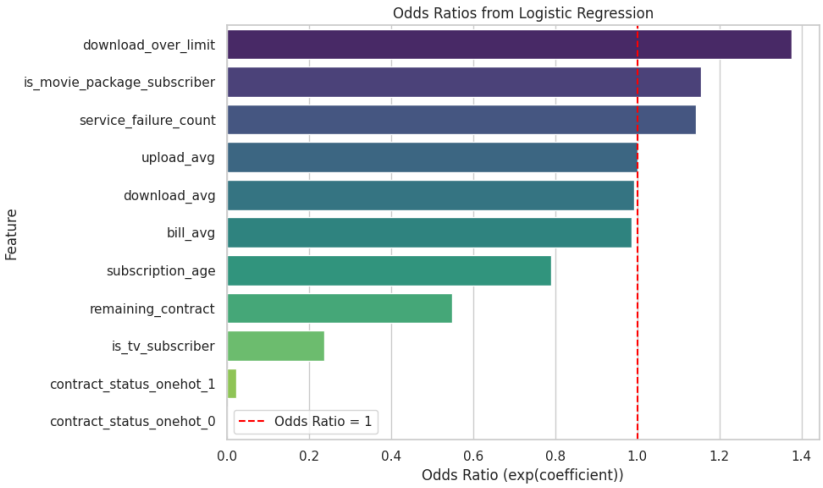
**Figure 25**: reconstruct one hot encoding and find the weights of each feature.

Create the weights_df, add the feature names we created and add the cofefficinets.

Then sort the DataFrame by impact strength to find the most influential features.

Sort by the absolute values.

Print the intercept and display the weights.




Intercept: 7.923886991129793

| | feature | coefficient | odds_ratio |
|---|---|---|---|
| 9 | contract_status_onehot_0 | -6.946797 | 0.001 |
| 10 | contract_status_onehot_1 | -3.795113 | 0.022 |
| 0 | is_tv_subscriber | -1.439536 | 0.237 |
| 4 | remaining_contract | -0.600664 | 0.548 |
| 8 | download_over_limit | 0.318355 | 1.375 |
| 2 | subscription_age | -0.234848 | 0.791 |
| 1 | is_movie_package_subscriber | 0.143752 | 1.155 |
| 5 | service_failure_count | 0.132687 | 1.142 |
| 3 | bill_avg | -0.014735 | 0.985 |
| 6 | download_avg | -0.008355 | 0.992 |
| 7 | upload_avg | 0.000946 | 1.001 |

# Biggest Churn Odds:

**Figure 26**: Odds ratios of features and a graph for visualization.

## Contract Status:

**contract_status_onehot_0 (active contract):** This feature has the strongest negative impact on churn. Customers with an active contract are 99.9% less likely to churn compared to those with an expired contract.

**contract_status_onehot_1 (never had contract):** Customers who never had a contract are 97.8% less likely to churn compared to those with an expired contract.

## TV Subscription:

**is_tv_subscriber:** Having a TV subscription reduces churn risk by about 76.3%, indicating it may be a valuable retention feature.

## Remaining Contract Time:

**remaining_contract:** For each additional unit of contract time, the odds of churn decrease by about 45.2%.

## Download Over Limit

**download_over_limit:** Customers who exceed their download limit are 37.5% more likely to churn, potentially due to poor experience or billing dissatisfaction.

## Accuracy:

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

evaluator = MulticlassClassificationEvaluator(labelCol="churn", predictionCol="prediction",
metricName="accuracy")
accuracy = evaluator.evaluate(predictions_lr)
print(f"Accuracy: {accuracy:.3f}")
```

**Figure 27**: Accuracy metrics for logistic regression.

**Logistic Regression Accuracy:** **0.925**

This is a great accuracy and shows our model is accurately predicting correct results, and the odds are accurate as well.

## AUC Score:

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

auc_evaluator = BinaryClassificationEvaluator(labelCol="churn", rawPredictionCol="rawPrediction",
metricName="areaUnderROC")
auc = auc_evaluator.evaluate(predictions_lr)
print(f"AUC: {auc:.3f}")
```

**Figure 28**: AUC score for Logistic Regression

**AUC Score:** **0.959**

This is a great AUC score and means that the logistic regression model can distinguish churners from non-churners with high accuracy.

## Predictions and Confusion Matrix

```
from sklearn.metrics import classification_report

pdf = predictions_lr.select("churn", "prediction").toPandas()
print(classification_report(pdf["churn"], pdf["prediction"]))
```

**Figure 29**: Prep and build the confusion matrix for Logistic Regression.

```
predictions_lr.groupBy("churn", "prediction").count().show()

▸ (2) Spark Jobs

+-----+----------+-----+
|churn|prediction|count|
+-----+----------+-----+
|    1|       0.0| 1082|
|    0|       0.0| 9139|
|    1|       1.0|10894|
|    0|       1.0|  546|
+-----+----------+-----+
```

**Again, we will make predictions on our logistic regression model and group it by churn and prediction, to form a confusion matrix shown below:**

| Churn? | Precision | Recall | F1-Score | Support |
|---|---|---|---|---|
| **0** | 0.89 | 0.94 | 0.92 | 9685 |
| **1** | 0.95 | 0.91 | 0.93 | 11976 |

**Class 0 (Non-Churners):**

**Precision:** 89% — 89% of predicted non-churners were correctly identified by the model.

**Recall:** 94% — The model correctly identified 94% of all actual non-churners.

**F1-score:** 0.92 — Strong balance of precision and recall.

**Class 1 (Churners):**

**Precision:** 95% — 95% of predicted churners were correct.

**Recall:** 91% — The model correctly identified 91% of all actual churners.

**F1-score:** 0.93 — Excellent performance and high confidence in churn detection.

**Answer to Question 2: How can we segment customers and churn based on their usage patterns, subscription type and behavior? And which features have the most impact on churn?**

I clustered usage customers into three categories which were light usage, moderate, and heavy usage. The highest churn rates were on light usage so targeting these groups might eliminate churn.

Subscription type I segmented into three groups the clustering did not help so I will refer to the odds ratios to determine how likely someone is to churn. The biggest feature on churn was tv_subscription by "having a TV subscription reduces churn risk by about 76.3%, indicating it may be a valuable retention feature"(Logistic Regression Odds) So segmentation into tv subscription and movie subscription, targeting tv_subscription could be a valuable way to eliminate churn.

There are two other behavior types that affect churn which is download_over_limit and remaining_contract. Here are the two statistics:

- **remaining_contract:** For each additional unit of contract time, the odds of churn decrease by about 45.2%.
- **download_over_limit:** Customers who exceed their download limit are 37.5% more likely to churn, potentially due to poor experience or billing dissatisfaction.

These are two ways to segment customers. The company should be inclined to have someone buy a contract when signing new customers onto their service. Meaning the more time someone has a contract the more likely they are to stay with the same company/service.

When a customer downloads over their limit for internet, the odds are 37.7% more likely to churn so when signing new customers this internet service provider should run tests on how much they download and their usages to distance themselves from dissatisfaction.

****Continue to Conclusion****

# Conclusion and Findings:

Our analysis of ISP customer churn behavior reveals drivers of churn and provides strategies for segmentation and retention:

1.) **How do subscription age and internet usage affect churn?**

Subscription age and internet usage both show strong relationships with churn behavior. Newer customers and those with very low internet usage are significantly more likely to churn, indicating a lack of engagement or dissatisfaction early in the customer lifecycle. On the other hand, long-tenured and higher-usage customers exhibit more loyalty, suggesting that increasing product engagement may help improve retention.

2.) **How can we segment customers and churn based on usage patterns, subscription type, and behavior?**

Using unsupervised clustering and Random Forest models, we identified distinct customer segments by usage, billing, and subscription patterns. Light-usage and low-engagement customers (those not subscribed to TV services) were more prone to churn. Customers with active contracts or higher service bundles (TV + internet) showed much lower churn. Segmenting based on service engagement, contract type, and billing behavior allows for more targeted interventions, such as personalized offers or proactive outreach to high-risk groups.

3.) **What attributes have the largest impact on churn?**

Across multiple models, including logistic regression, random forest, and odds ratio analysis, several attributes consistently emerged as key churn predictors:

Remaining contract time: Customers with more time left on their contracts are less likely to churn.

TV subscription: Having a TV service significantly reduces churn risk (by ~76%).

Download over limit: Customers who exceed their data limits are over 35% more likely to churn.

Internet usage: Low-usage customers churn more frequently, possibly due to underutilization or dissatisfaction.

These attributes highlight the importance of bundling services and monitoring usage behavior to identify churn risks early.

### 4.) What is the average Customer Lifetime Value (CLV), and can we predict it?

We engineered a CLV feature using historical billing and contract data and attempted to predict it using both linear regression and random forest models. While the models achieved limited accuracy ($R^2$ values below 0.17), we learned that CLV is highly individualized and influenced by behaviors not fully captured in the available features. Despite low predictive power, the most informative variables included remaining contract time, service bundle types, and past usage behavior. With more granular or behavioral data (e.g., customer support interactions), CLV prediction could be improved.