Final Year Project Report

Full Unit – Final Report

Customer Churn Prediction: A Comparative Analysis of

Machine Learning Algorithms

Abdulahi Said

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Dr Anand Subramoney

Department of Computer Science

Royal Holloway, University of London

April 24, 2025

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 13942

Student Name: Abdulahi Said

Date of Submission: 24/04/2025

VIDEO LINK: https://youtu.be/DacHd-H8fwY

Table of Contents

Abstract

When clients discontinue using a service, customer churn becomes a critical issue across many industries, especially for subscription-based businesses. In such sectors, customer satisfaction and retention are vital to ensuring steady revenue streams. With intense competition, every lost customer can represent a direct gain for competitors [1]. On average, businesses lose between 10% and 25% of their customer base annually due to churn, with some industries, such as wholesale, experiencing rates as high as 56% [2][3].

High churn rates undermine a business's ability to establish and maintain a loyal customer base, essential for long-term profitability. Studies have shown that retaining customers is much more profitable and sustainable [5]. This is because customer retention requires fewer resources compared to acquiring new customers and because loyal customers tend to spend more over time. As a result, businesses that focus on strengthening relationships with existing clients can boost their profitability while ensuring steady growth.

To address this known issue, this project aims to conduct a comparative analysis of various machine learning (**ML**) algorithms designed to predict customer churn effectively. These algorithms include K-Nearest Neighbours (**KNN**), Decision Trees (**DT**), Extreme Learning Machines (**ELM**), and Random Forest **(RF**). A significant challenge in churn prediction is class imbalance. This occurs as the number of churners is outnumbered by customers that remain. This can lead to biased model predictions, increasing the number of false positives and false negatives. Such inaccuracies have damaging repercussions on businesses as they may waste financial and marketing resources targeting the wrong customers while overlooking actual customers who would leave. To deal with these issues, I will implement robust data preprocessing techniques, such as handling missing values and scaling features. Additionally, I will explore ensemble methods to improve model accuracy and generalisation [6].

Furthermore, my project will apply these ML models to a benchmark dataset. Each algorithm's performance will be measured using various measures, including precision, recall**, F1** and **F2 score**, thus ensuring comprehensive model assessment. To enhance accessibility for non-technical users, I will develop a GUI that allows users to visualise the performance of selected ML models, making it more user-friendly and intuitive.

Project Specification

Aims: To implement and compare on benchmark data sets various machine learning algorithms

Background: Machine learning allows us to write computer programs to solve many complex problems: instead of solving a problem directly, the program can learn to solve a class of problems given a training set produced by a teacher. This project will involve implementing a range of machine learning algorithms, from the simplest to sophisticated, and studying their empirical performance using methods such as cross-validation and ROC analysis. In this project you will learn valuable skills prized by employers using data mining.

Early Deliverables

1. You will implement simple machine learning algorithms such as nearest neighbours and decision trees.

2. They will be tested using simple artificial data sets.

3. Report: a description of 1-nearest neighbour and k-nearest neighbours algorithms, with different strategies for breaking the ties;

4. Report: a description of decision trees using different measures of uniformity.

Final Deliverables

1. May include nearest neighbours using kernels and multi-class support vector machines.

2. The algorithms will be studied empirically on benchmark data sets such as those available from the UCI data repository and the Delve repository.

3. For many of these data set judicious preprocessing (including normalisation of attributes or examples) will be essential.

4. The performance of all algorithms will be explored using a hold-out test set and cross-validation.

5. The overall program will have a full object-oriented design, with a full implementation life cycle using modern software engineering principles.

6. Ideally, it will have a graphical user interface.

7. The report will describe: the theory behind the algorithms.

8. The report will describe: the implementation issues necessary to apply the theory.

9. The report will describe: the software engineering process involved in generating your software.

10. The report will describe: computational experiments with different data sets and parameters.

Suggested Extensions

- Modifications of known algorithms.
- Dealing with machine learning problems with asymmetrical errors (such as spam detection) and ROC analysis.
- Nontrivial adaptations of known algorithms to applications in a specific area, such as medical diagnosis or option pricing.
- Exploring the cross-validation procedure, in particular the leave-one out procedure. What is the optional number of folds?
- Comparative study of different strategies of reducing multi-class classifiers to binary classifiers (such as one-against-one, one-against-the-rest, coding-based).

Reading

- Trevor Hastie, Robert Tibshirani, and Jerome Friedman. The Elements of Statistical Learning. Second edition. Springer, New York, 2009.
- Tom M. Mitchell. Machine Learning. McGrow-Hill, New York, 1997.
- Vladimir N. Vapnik. The Nature of Statistical Learning Theory. Second edition. Springer, New York, 2000.
- Vladimir N. Vapnik. Statistical Learning Theory. Wiley, New York, 1998.
- Seth Hettich and Steven D. Bay. The UCI KDD Archive. University of California, Department of Information and Computer Science, Irvine, CA, 1999,

Chapter 1: **Introduction to Key Concepts**

1.1 What is Machine Learning?

Machine Learning is a subset of artificial intelligence driven by algorithms designed to identify patterns in data. These patterns allow the algorithm to make predictions without requiring explicit programming for each task, whether it is predicting if an image is a cat or predicting stock prices. The adaptability of machine learning makes it a powerful tool in data-driven applications, helping to tackle many challenges across various industries.

Machine Learning can be classified into three categories:

1. **Supervised Learning**: Uses labelled data that contains a feature and target to learn patterns and relationships. Once trained, the model can generalise from the learned patterns to predict the target labels of unseen data. For instance, predicting churn is a supervised problem, where historical data on whether customers have churned or not is used to train the model to make accurate predictions.

2. **Unsupervised Learning:** Utilises data that doesn’t have pre-existing features and label mappings to discover patterns and relationships. An example of usage would be clustering customers based on similar behaviours.

3. **Reinforcement Learning** is quite different from supervised and unsupervised learning. It involves learning through trial and error by interacting with an environment to achieve the most optimal results through sequential decision-making and actions.

Given that my project is customer churn prediction, I will implement supervised learning using features such as tenure, monthly, charges, services and payment methods to predict whether a customer would churn. This would then allow businesses to take proactive measures for customers retention.

1.2 Classification vs Regression

Depending on the predicted label, most machine-learning problems can be classified into regression or classification tasks. Regression is a type of supervised learning that predicts continuous numerical values, such as the temperature or house prices. On the other hand, classification predicts using discrete categories or labels such as ‘yes’ and ‘no’.

Customer churn prediction would be classified as a classification problem. This is because the goal is to assign the two discrete labels churned (1) and not churned (0). It is not a regression problem, as it does not involve predicting a continuous value representing the likelihood of churn. Instead, it focuses on categorising customers based on their likelihood of discontinuing the service, making classification the appropriate approach for this project.

1.3 Binary vs. Multi-Class Classification

In machine learning, classification problems can be categorised into binary and multi-class. Binary classification is where only two possible class labels can be assigned, such as ‘churned’ and ‘not churned’. Multi-class classification involves predicting the label of a data point from three or more possible categories. An extension of the churn example for multi-class classification could be adding different likelihood risks of churn such as “none”, “low”,’ medium’ and “high”.

In my project, the churn prediction would be initially treated as a binary classification problem, where the goal is to categorise each customer into one of two categories: those likely to churn and those not likely to churn. This approach simplifies the problem, providing a good foundation for me to carry out experimentation and analysis. There is also the flexibility to later expand into a multi-class classification by adding different levels of churn risk.

1.4 Dataset I will be using

The Dataset that I have chosen to use is a Telecoms customer churn dataset from IMB[7]. This dataset contains 7043 customers with 21 features capturing services used by customers and subscription details.

1.4.1 Key Features:

- **Customer Demographics**: Information such as gender , senior citizen status and whether they have dependants.
- **Service Details**: Features like whether the customer has phone service, multiple lines, or internet service and whether the customer uses other services such as security and streaming services.
- **Contract Information**: The type of contract (month-to-month, one-year, two-year), payment method, and whether paperless billing is enabled.
- **Charges and Tenure**:
- **Tenure**: The number of months a customer has been with the telecoms service.
- **Monthly Charges**: The monthly cost of the services used
- **Total Charges**: The cumulative charges over the entire tenure.

1.4.2 Label

The label in this dataset would be churn, a binary label indicating whether a customer has churned “yes” or hasn’t churned “no”. I have converted these into 0 and 1, 0 representing not churned and 1 representing churn.

1.4.3 Why this dataset?

I selected this dataset because it was highly rated on Kaggle for its quality and because IBM, a well-known leader in telecommunications services, produced it. Since it simulates real-world churn scenarios, it makes it a suitable choice for machine-learning models.

Chapter 2: **Performance Evaluation**

2.1 Introduction of Model Evaluators

Evaluating models is a crucial part of this project. The primary objective is to compare a range of machine learning algorithms to determine the most effective for predicting churn. It is also key as it informs whether the ML model generalises well to unseen data and the model's capabilities to classify whether a customer is a churner correctly.

As I have established before, this is a classification problem, and the choice of metrics is important. We cannot use mean squared error (MSE) or R-squared as the label is not continuous. Metrics such as precision, accuracy, recall and F1-score are more suitable for classification and give a detailed insight into the model's performance. For example, a high recall score ensures that most actual churners are identified, which is key if companies want to target those customers with retention strategies. Similarly, a high precision score minimises the risk of misclassifying non-churners as churners, avoiding unnecessary usage of resources and intervention to convince customers to stay.

2.2 Evaluation Metrics explained

- **True Positive (TP)**: A churner correctly identified as a churner.

- **False Positive (FP)**: A non-churner incorrectly identified as a churner.

- **False Negative (FN)**: A churner incorrectly identified as a non-churner.

- **True Negative (TN)**: A non-churner correctly identified as a non-churner.

2.2.1 Accuracy

The most common metric is accuracy; it measures a model's number of correct predictions divided by the total number of predictions. While this metric is simple to use, it can be misleading as it does not account for the number of false negatives and false positives. Since the churn dataset is imbalanced, where the majority of customers do not churn, as compared to churning, the model could have a high accuracy score based on predicting the majority class, rendering this metric less useful in identifying actual churners(True negative).

2.2.2 Precision

Precision is the proportion of true positives identified by the model out of all instances it predicted as positive. In relation to our churn prediction, precision measures the number of correctly identified churners (true positives) divided by the total customers predicted as churners (true positives + false positives).
Precision = (True Positives) / (True Positives + False Positives)

A high precision score indicates that the model minimises false positives, ensuring resources are focused on actual churners rather than misclassified non-churners.

2.2.3 Recall

Recall, also known as sensitivity, measures a model's ability to identify the number of true positives in a dataset. In relation to our churn prediction, this would be the proportion of correctly identified churners out of all instances predicted as churners.

Recall = (True Positives) / (False Negatives + True Positives)

The recall score must be high so that at risk customers can be identified and targeted with retention strategies. A low recall score means that many actual churners are missed (false negative), causing lost revenue as these customers would leave without any intervention.

2.2.4 F1-Score

A metric that combines recall and precision, which is used in classification problems. It provides a balanced assessment of a model's performance as it considers both recall and precision. It is very useful when datasets are imbalanced, which they are in this case.

F1-Score = 2 × (Precision * Recall) / (Precision + Recall)

A high F1 score shows that the model effectively identifies churners and keeps false positives low. This is important since, in a business environment, misclassification can affect financial and resource consequences.

2.2.5 F2 -Score

A variation of the F1-score but places a greater emphasis on recall than precision. It’s useful when identifying actual churners is more important than avoiding false positives.

- β2 = 2
- F2-Score = (1+β2) × ((Precision × Recall) / ((β2×Precision) + Recall))

F2-Score has limitations so I would use it in conjunction with all the other metrics

Chapter 3: **Explorative Data Analysis and Data Preprocessing**

Explorative Data Analysis (EDA) and data preprocessing are critical steps in the machine learning pipeline. Explorative data analysis provides a deeper understanding of the dataset by visualising patterns through graphs and statistical summaries, detecting anomalies. The deeper understanding gained from EDA aids in parameter tuning and feature selection, guiding our overall modelling approach and results [8].

Data preprocessing essentially transforms the raw data into clean data by handling missing data values and encoding data in a way that is suitable for the ML model and normalising features. These steps ensure that the model is trained on quality and reliable data, thus enhancing the accuracy and reliability of our results [9].

3.1 Explorative Data Analysis on Churn Dataset

The dataset I used in this project, described in section 2.4, is the Telco Customer Churn dataset. This section focuses on the insights I gained from carrying out exploratory data analysis.

To gain an initial understanding of my data, I converted it into a data frame and used the command .head(). This provided an overview of the structure, including feature names, data types, and sample values, ensuring familiarity with the dataset's layout. Since missing data is a critical aspect, I utilised a heatmap to output which attributes have missing data as shown in figure1.

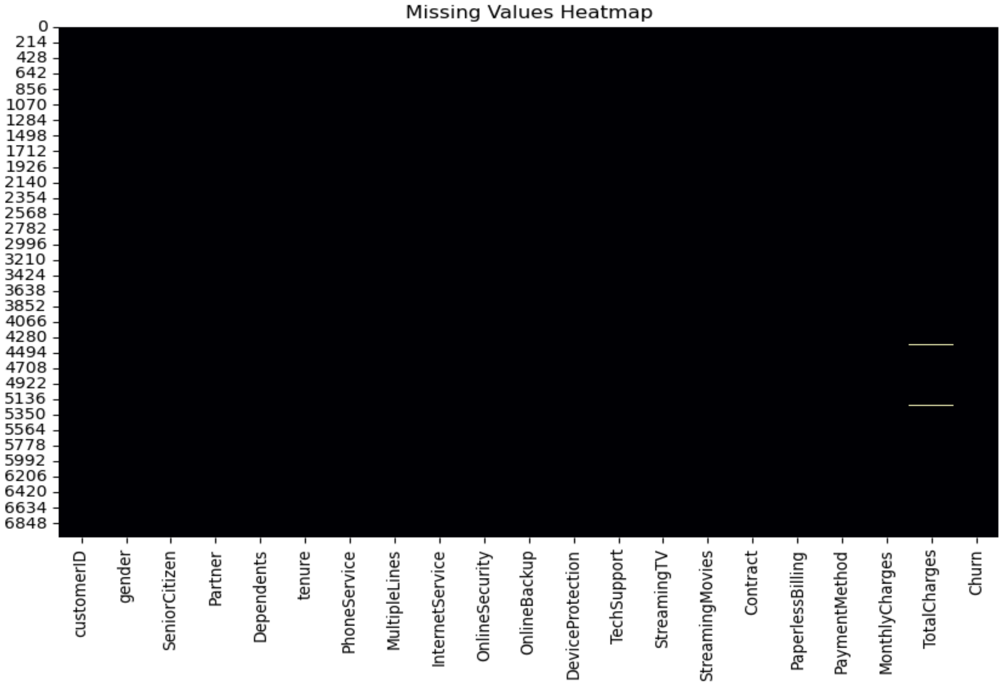


Figure 1. Heatmap of missing Values in the Dataset

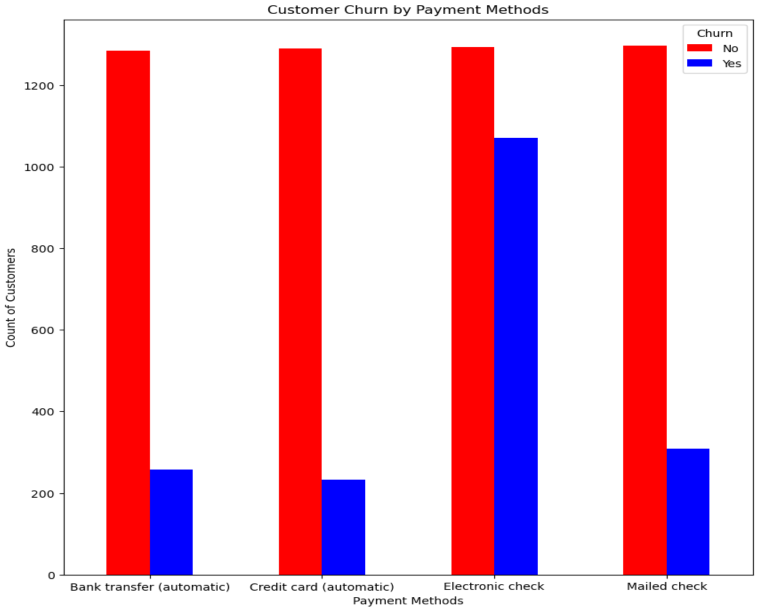
The heatmap revealed that total charges had missing data, and upon closer examination, eleven customers had missing data for the Total charges column. This insight was essential for data preprocessing phase.

3.1.1 Insights from Visualising Relationship

I began investigating the relationships between different features through various visual plots that include, histograms, boxplots and bar charts.

1. Payment Methods and Churn: The bar charts showed that customers using electronic checks have a higher churn rate than customers using other methods, such as credit cards and bank transfers. This could imply customer dissatisfaction with certain payment methods.

Figure 2. Customer Churn by payments Bar Chart



2. **Impact of Monthly Charges**: A boxplot of Monthly Charges against Churn revealed customers with higher monthly charges show a greater likelihood of churning, indicated by the higher median and narrower spread of charges for churners compared to non-churners in figure 3. This suggests highly to me that monthly charge is a key feature.

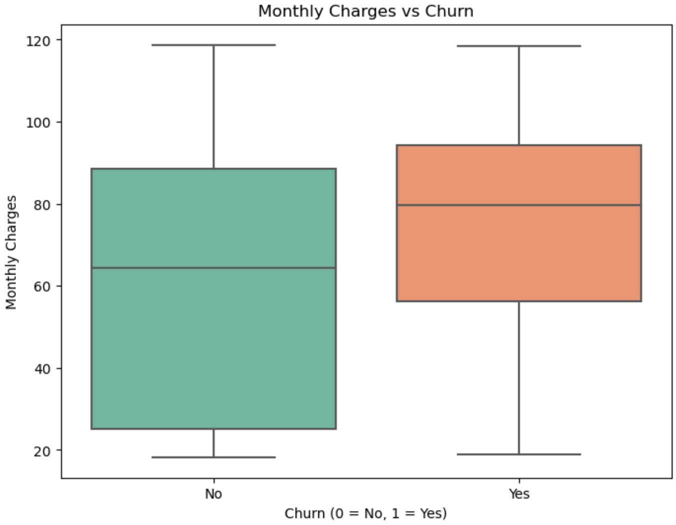


Figure 3. A box plot of Monthly Charges vs Churn

Figure 4.

3. **Correlation Matrix:** Figure 4 displays a correlation matrix of the numerical features of the dataset. From Figure 4 we can infer a strong positive correlation between Total Charges and Tenure. Also, monthly charges and tenure show a weak positive correlation (0.247), while

senior citizens have a minimal correlation with other features, indicating limited predictive benefits for churn.

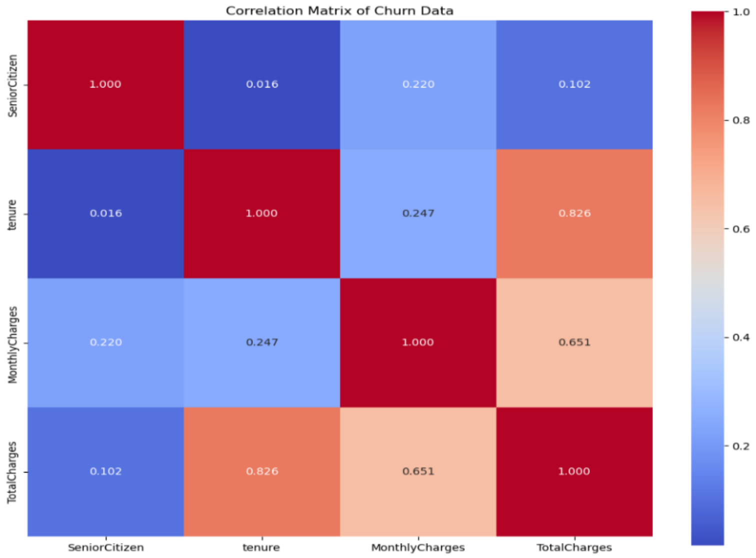


Figure 5. Correlation Matrix of Churn Dataset

3.2 Data Pre-Processing

3.2.1 Handling Missing Values

During my exploratory data analysis, I discovered that the Total Charges column had 11 missing values. My initial decision was to delete these 11 rows from the dataset, given that all those customers haven't churned and also given the class imbalance. I deemed it would have been fine. However, upon closer inspection, I realised that these rows included valid Monthly Charges and Tenure values, which could be used to manually calculate the missing Total Charges values. Since Tenure represents the number of months a customer has been with the service provider, and Monthly Charges reflect the recurring monthly cost, the missing Total Charges values were calculated using the formula: Total Charges = Monthly Charges * Tenure. By utilising this relationship, this ensured that no data was wasted and maintained the integrity of the dataset.

3.2.2 Encoding Categorical Features

Many machine learning models struggle to process categorical features. Since my dataset includes features like gender and payment methods, I applied one-hot encoding to these categorical variables. One-hot encoding was chosen because it prevents ordinal relationships from forming and enables each category to be treated independently, thus avoiding misleading relationships between them. This is very important in linear and distance-based models, where ordinal relationships can affect the model's performance negatively.

3.2.3 Feature Scaling

Feature scaling is a preprocessing technique that scales numerical data to a similar scale. It is crucial in models that rely on distance or gradient-based calculations to fit and predict. By ensuring that all features contribute proportionally, scaling prevents those with more extensive ranges from dominating the learning process. I applied both **Min-Max scaling** and **Z-score normalisation** during preprocessing. Z-score was primarily used throughout the project, and Min-Max scaling was used for experiments on how scaling affects KNN performance.

Chapter 4: **K-Nearest Neighbours (KNN) model**

4.1 Background Theory

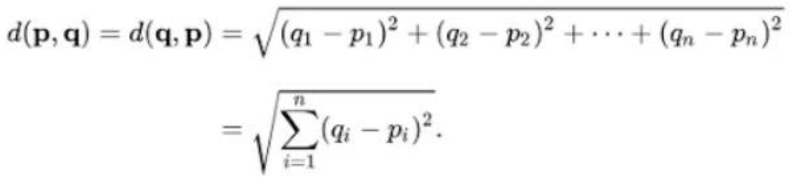
The K-Nearest Neighbour (KNN) algorithm is a supervised learning method widely used for classification tasks due to its simplicity and effectiveness. It utilises the nearest neighbour of the given data point to determine the class. The parameter K represents the number of neighbours considered, and the final prediction would be based on a majority vote. In the case of a tie, various methods can be used to resolve it, such as random voting. However, after running KNN extensively on the dataset utilising a range of k from 1-50, I encountered 0 ties, and therefore, I did not investigate this issue extensively. However, for precaution in the event of a tie, the first class among the tied options is selected.

Furthermore, KNN is a lazy learner, and no real training is required, as distance computations between data points are carried out during the prediction phase. This makes KNN a memory-intensive algorithm, especially with large, high-dimensional datasets [10].

4.1.1 How are distances between datapoints calculated?

In KNN, the classification of an unknown data point heavily depends on the distance between labelled data points and the selected k parameter. This makes distance a critical step, but how do we compute the distances? There are several distance metrics, each suited to different purposes:

1. Euclidean distance:



$$d(\mathbf{p},\mathbf{q}) = d(\mathbf{q},\mathbf{p}) = \sqrt{(q_1-p_1)^2+(q_2-p_2)^2+\cdots+(q_n-p_n)^2}$$
$$= \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}.$$

Figure 6. Euclidean distance equation

This equation calculates the straight line distance between 2 data points by summing the difference between the two coordinates and taking the square root.

2. Manhattan Distance: It computes the sum of the absolute difference between the coordinates of the data points.

   d = Σ | p_i – q_i |

   P and Q are vectors

   i represents the ith element of the vector

   Manhattan distance is generally used on gird paths

3. Cosine Similarity measures the angular difference between data points and is useful when the dataset's dimensionality is high, although it's not an actual distance metric.

   Cosine Similarity = (x · y) / (||x|| × ||y||)

4.2 Implementation of KNN

I decided to implement KNN from scratch without using libraries such as SKLearn. The main goal was to test my understanding of the theory and see how well it performs in customer churn classification. After brainstorming, my initial implementation was to create a KNN class that encapsulates the algorithm's key functionalities.

The initialisation function handled the K parameter and utilised safeguards to prevent erroneous entries, such as negative numbers. If not specified, neighbours were defaulted to 1. Since there is no real training, the fit method stores the corresponding training data: the features (X_train) and the labels (y_train). The next stage was determining the k-nearest neighbours for a given data point and selecting an appropriate distance metric. I chose the **Euclidean distance metric**, which is widely used and suitable for numerical data. It calculates the straight-line distance between two points in the feature space, defined in Figure 6.

To optimise the distance computations, I utilised NumPy's vectorised operations, significantly improving calculation speed compared to looping through the entire dataset. This approach ensured efficient processing, even with larger datasets.

The code for calculating Euclidean distance is shown below:

```
    def euclidean_distance(self,p1,p2):
        return np.sqrt(np.sum((np.array(p1) - np.array(p2)) ** 2))
```

When the KNN model does prediction, the algorithm calculates the distance between the test data point and every data point in the training set. This would involve iterating through the training dataset and computing the Euclidean distance for each pair. The issue with this is that sorting becomes timely, especially since my dataset is over 7000. I initially used a general insertion sort function I created but quickly realised that this method was inefficient as the time complexity of insertion sort is **$O(n^2)$,** meaning that, in the worst case, the time taken to sort a list is proportional to the square of the number of elements in the dataset. To address this issue, I utilised python’s heapq library, which efficiently identifies k smallest distances without sorting the whole list. This optimisation significantly reduced the run time of my KNN model.

Code behind getting K-nearest neighbour:

```
def get_nearest_neighbour(self, test_sample):
    # Checks if we set K > size of sample and deals with it accordingly
    k = min(self.n_neighbours, len(self.X_train))

    # Calculate all distances without sorting
    distances = []
    for train_sample, label in zip(self.X_train, self.y_train):
        distance = self.euclidean_distance(train_sample, test_sample)
        distances.append((distance, label))

    # Use heapq to get the k smallest distances directly
    k_nearest_neighbours = heapq.nsmallest(k, distances, key=lambda x:
x[0])
    return k_nearest_neighbours
```

4.3 Running my KNN model

After implementing the KNN model and testing it utilising TDD development and unit testing, I then ran it on my churn dataset. To evaluate the model results I utilised my evaluator class that outputs accuracy, precision, recall, F1-score and F2-score. The aim was to find the optimal value of k, ensuring the best possible results.

4.3.1 My setup for investigating k-values

To evaluate the model, I split the data into test and train, with 20% of the data being for testing and the other 80% for training. I then created a function to test k from range 1-50, storing results in an array, which would then be plotted as a line graph. This would make investigating the optimal K value much easier.

4.3.2 Model Results across different K-values:

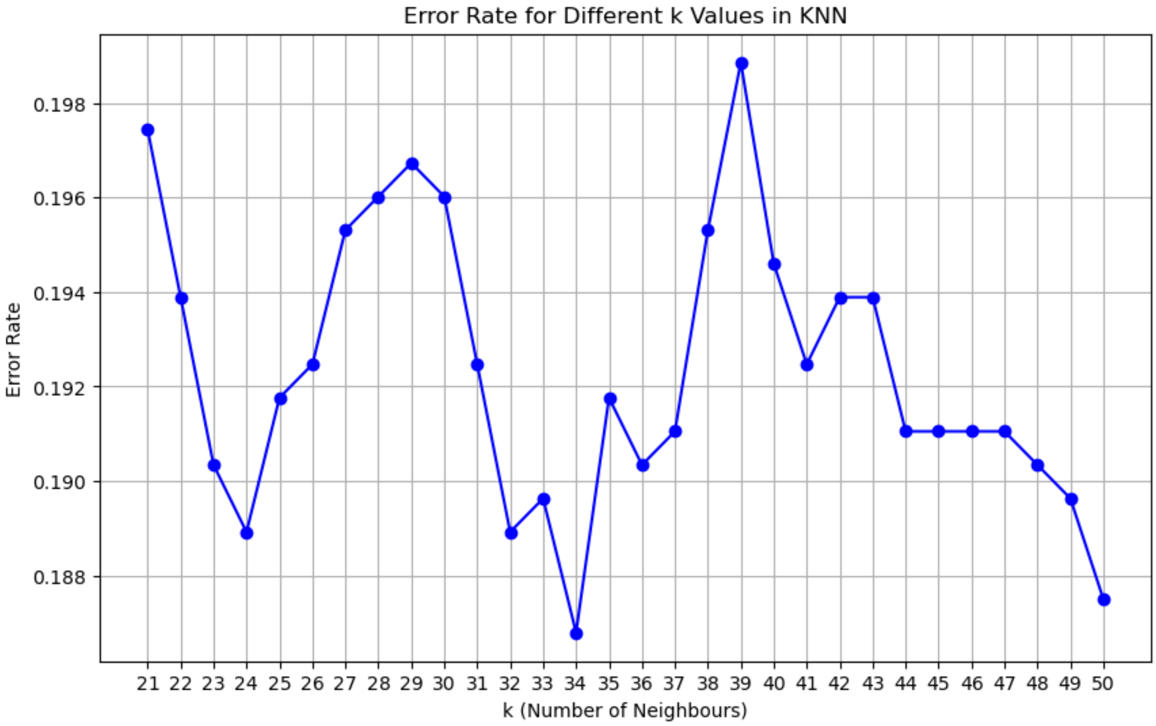


Figure 7. Error rate for different K-values of KNN model

While you cannot see from graph the error rates of k from range 1-20, the error rate was higher than range k 21-50 . K values in the range 1-20 exhibited the highest error rates but were gradually declining as k increased, and then there was a noticeable drop at k = 21. Its likely that very small values of k performed poorly as they are much more likely to capture noise and any outliers in the dataset that would lead to more unreliable predictions.

For k values greater than 20, the error rate appeared to fluctuate significantly in the graph; however, these fluctuations were within very small margins, in the range of 3 decimal places. This is why the graph looks very volatile, but in reality, it is stable. Looking at the graph, the optimal k was when k = 34, as it had the lowest error rate. For K values greater than 34 it continued to fluctuate.

4.3.3 Optimal K value comparison using different Normalisation.

After identifying the optimal k-value as k =3 4k, I decided to investigate the impact of different normalisation techniques on the model's performance. Normalisation is essential for distance-based algorithms like KNN, ensuring that features with larger values do not disproportionately influence the distance computation [10].

For this comparison, I used two widely adopted techniques: Min-Max Scaling and Z-Score Scaling**.**

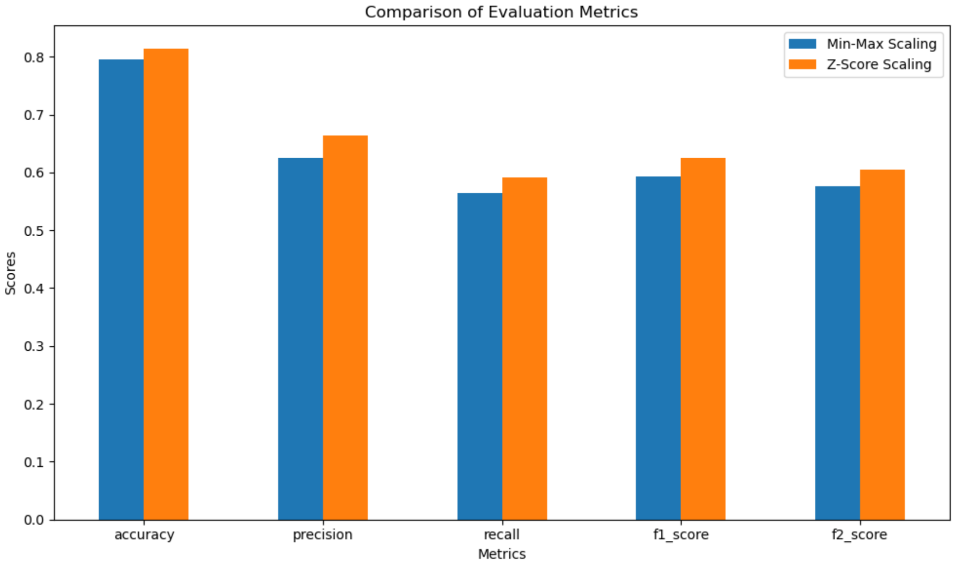


Figure 8. K=34, knn model with different Normalisation method

The performance of k = 34 was assessed using my evaluator metric class, and Figure 8 displays the performance between the two normalisation methods.

Observations:

- Accuracy: Both normalisation methods had a fairly high accuracy, but z-score performed slightly better than min max.
- Precision and recall: Z-score consistently achieved higher precision and recall than min max. Thus, indicating it manages the feature distribution of the churn dataset more effectively
- F1/F2 Score: z-score demonstrated a slightly higher score than min-max normalisation.

This investigation revealed that normalisation techniques affect KNN model performance, but not significantly.

Despite these variations, the KNN model demonstrated decent overall performance. These results for precision, recall, and f1-score are understandable given the imbalanced nature of the dataset and the inherent characteristics of KNN as a lazy learner.

Chapter 5: **Decision Tree Model**

5.1 Background Theory

5.1.1 What is a decision tree?

A decision tree is a supervised learning algorithm that can be used for both classification and regression. Structurally, it resembles a tree structure consisting of a root node, which is the tree's base, and branching from it is a hierarchy of decision nodes terminated by leaf nodes. Leaf nodes are nodes with no children.

A decision tree is a supervised learning algorithm that can be used for both classification and regression. Structurally, it resembles a tree consisting of a root node, which is the tree's base. Branching from it is a hierarchy of decision nodes terminated by leaf nodes. Leaf nodes have no further partitions, meaning they have no children. Also, each decision node represents a splitting point where the data is divided into subsets based on a feature value or condition. The criteria for splitting could be vast, from whether monthly charges exceed a certain number to categorical questions such as whether the customers have a dependant. This branching process would continue until partitions of the data are pure or another stopping criterion is met.

Purity refers to how homogeneous the data is after splitting. A data partitioning is considered pure if and only if all the data points belong to that class. For example, in my churn prediction, a subset that contains data points of those with higher monthly charges than 27 could be considered pure. Decision trees aim to decide the splitting of data by maximising purity.

5.1.2 Different Measures of purity

Many different measures of purity decide splits and I will be focusing on three:

1. **Gini Impurity**: returns the likelihood of new random data being misclassified if given a random label from the possible classes. Its result Ranges from 0-0.5

   0 value indicates a perfect purity (each datapoint belongs to single class) and 0.5 means maximum impurity (the datapoints are split equally between the classes).

$$Gini = 1 - \sum_{i=1}^{C} (p_i)^2$$

Figure 9. Gini impurity equation

2. **Entropy :**

   Pi is the proportion of data points, in this case, customers that belong to the same class.

$$E(S) = \sum_{i=1}^{c} - p_i \log_2 p_i$$

   0 Indicates a completely pure subset (all customers belong to the same class i.e churners or not churners

Figure 10.

3. **Classification error:** It measures the proportion of incorrectly classified data points in a subset. A very simple metric and equation is: Classification Error =1 − max(pi).

5.2 Implementation of Decision Trees

Like my KNN model, I implemented it from scratch and used the background theory I learnt to develop it. My initial implementation was to have a single class that would handle the basic tree construction with different uniformity measures and then predict unseen data. However, looking at future models to be developed and extensions of the decision trees model, such as implementing ensemble methods like random forests . I decided to opt for a more modular approach by developing two classes: Node and Decision Tree. There were some challenges, but it was worth it for the flexibility this class would provide for later usage.

Why only two classes? The **Node** class is designed to represent the building blocks of the decision tree.

```
class Node:
       def __init__(self, feature=None, threshold=None, left=None,
right=None, *, value=None):

        """
        Initialises a node in the decision tree.

        Parameters:
        - feature: Index of the feature used for splitting.
        - threshold: Threshold value for the split.
        - left: Left child node.
        - right: Right child node.
        - value: Class label if the node is a leaf.
        """
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

    def is_leaf_node(self):
        return self.value is not None
```

This initialisation function contains the essential attributes and has a leaf function to determine if the node has no children, thus simplifying tree traversal.
Developing this node class allowed it to be reusable and flexible. It also made it much easier to test and debug when the tree-building process occurred.

5.2.1 Decision Tree class

Having developed the node class, I then moved on to developing the main class(decision tree).The decision tree class's purpose is to construct the tree, train it using the churn dataset, and make predictions. The fit method was the starting point of the tree construction and would carry out a recursive call to the grow function, which I will discuss.

Challenges and notable methods for decision tree class:

1. Handling the splitting criteria: Initially, I only implemented gini impurity as the default for deciding when to split. However, I realised that if I were to investigate how different uniformity measures affect decision trees, I would have to make them modular. To handle this, I created a uniformity measure parameter that would take 3 of the different ones, as discussed, and developed a flexible impurity function to calculate impurity based on the chosen metric.

   Code for impurity, utilising the equations for different uniformity measurements:

   ```
    def impurity(self, y):
   """
   Calculate node impurity based on the chosen uniformity_measure.
   """
   proportions = np.bincount(y) / len(y) # p(X)

   if self.uniformity_measure == "gini":
       return 1 - np.sum(proportions ** 2)
   elif self.uniformity_measure == "entropy":
       return -np.sum([p * np.log2(p) for p in proportions if p> 0])
   elif self.uniformity_measure == "error":
       return 1 - np.max(proportions)
   ```

2. Deciding on the stopping criteria and how to manage it was an issue. In my first basic implementation, the tree would grow excessively, leading to overfitting, especially since I have about 30+ features after encoding. This would lead to overfitting, and to resolve this, I added parameters such as max_depth and min_samples_split; this would allow me to control the complexity of the tree. This would also allow me to carry out parameter tuning to allow me to get the best performance results from decision trees model.

3. Traversing the tree is key in the decision tree because to predict new data points, I would need to traverse the tree from the root node to the leaf node. There are several tree traversal methods, such as postorder, inorder and preorder, but for predictions of decision trees, the traversal is very specific. The feature values and thresholds guide the traversal in decision trees as each node, the feature value of the datapoint, would be compared against the threshold value. I recursively checked the feature values against thresholds and moved left and right in the tree until a leaf node was reached. The traversal was made more straightforward due to the design of my node class.

4. To ensure the model's robustness, I implemented verification and edge case handling. This includes addressing scenarios where empty splits are selected, causing errors and added complexity. Furthermore, implementing practical functions such as saving and loading models for my customer churn project was beneficial. By utilising Python's Pickle library, the trained model could be reused and computational resources saved.

5.3 Results and Running of my Decision Tree Model

I ran my decision tree on the churn dataset and saw its results. Before running, I split the data using 20% for testing and 80% for training.

Gini Impurity

- Performance Metrics: Accuracy: 0.793 (79.3%), Precision: 0.620 (62.0%), Recall: 0.618 (61.8%), F1 Score: 0.619 (61.9%), F2 Score: 0.619 (61.9%)

- Cross-Validation: Accuracy: 0.783 (78.3%), Precision: 0.618 (61.8%), Recall: 0.495 (49.5%), F1 Score: 0.544 (54.4%), F2 Score: 0.513 (51.3%)

Entropy

- Performance metrics: Accuracy: 0.789 (78.9%), Precision: 0.616 (61.6%), Recall: 0.592 (59.2%), F1 Score: 0.604 (60.4%), F2 Score: 0.597 (59.7%).

- Cross–Validation: Accuracy: 0.777 (77.7%), Precision: 0.648 (64.8%), Recall: 0.349 (34.9%), F1 Score: 0.452 (45.2%), F2 Score: 0.384 (38.4%)

Classification error

- Performance Metrics: Accuracy: 0.773 (77.3%), Precision: 0.677 (67.7%), Recall: 0.317 (31.7%), F1 Score: 0.432 (43.2%), F2 Score: 0.355 (35.5%). s

- Cross-validation: Accuracy: 0.782 (78.2%), Precision: 0.615 (61.5%), Recall: 0.494 (49.4%), F1 Score: 0.543 (54.3%), F2 Score: 0.511 (51.1%)

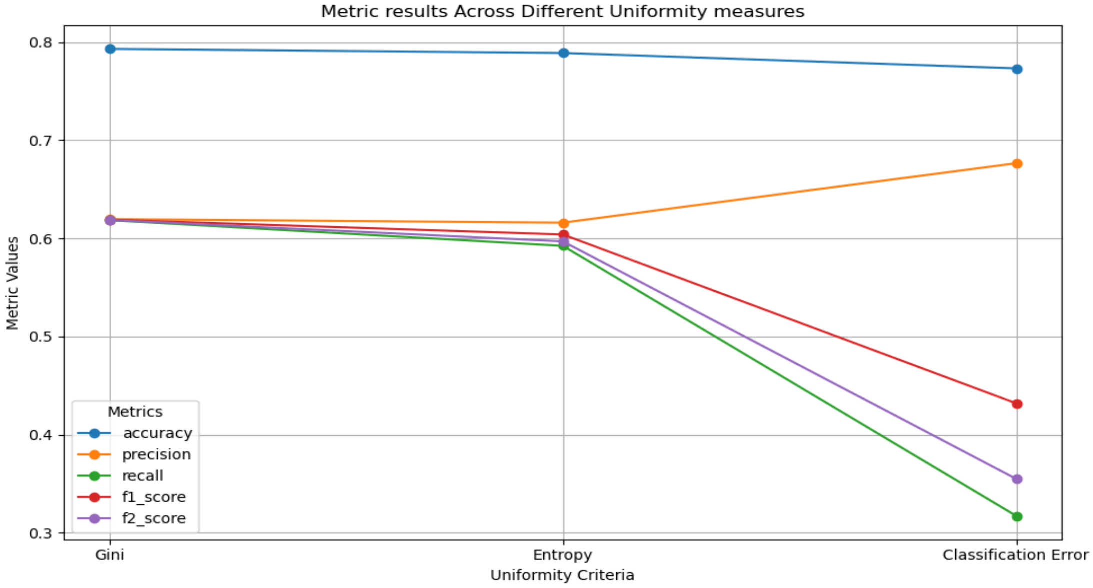


Figure 11. Results for different uniformity measures.

5.3.1 Analysis of results:

The Gini Impurity achieved the highest overall performance, demonstrating its effectiveness at identifying true positives and avoiding false positives.

Gini Impurity was the best split criterion, providing the best balance between the two classes, with an accuracy of 79.3%, precision of 62.0%, and recall of 61.8%, making it the most reliable for this dataset. Entropy performed slightly worse, with an accuracy of 78.9% and a lower recall of 59.2%, leading to slightly lower F1 and F2 scores. Classification Error performed the worst, with a very low recall of 31.7%, an accuracy of 77.3%, and a precision of 67.7%, indicating it is unsuitable for imbalanced datasets. In comparison, Gini Impurity was the most effective criterion, followed by Entropy, while Classification Error was the least effective.

To assess the model's robustness, I carried out a five-fold cross-validation of the model. The results conclude that Gini impurity maintained the most consistent performance despite a slight drop in recall (49.5%), which shows good generalisability. On the other hand, **Entropy** and **Classification Error** had a more noticeable decline in performance; the precision remained relatively consistent as its test set, but a significant drop in recall performance caused a reduction in the F1 and F2 scores.

To conclude, Gini Impurity proved to be the most effective and stable impurity measure, given its performance. However, it's not the best model to deal with imbalance datasets, and there exist other methods to boost the recall and F2-scores, which I will delve into in later chapters.

Chapter 6: **Logistic Regression**

6.1 Why Logistic Regression

As part of the project specification, K- Nearest Neighbours and Decision Trees classifiers were compulsory. While they were helpful as a gateway into the various types of Machine learning models (Trees, Linear), they are not the most well-suited models for churn predictions. Decision Trees can overfit easily on noisy or imbalanced data, and KNN tends to struggle with high-dimensional or sparse. Both are evident in my dataset, being imbalanced and high dimensional.

In contrast, Logistic Regression (LR) is a widely recognised ML model for classification problems such as the churn prediction problem I am doing. It is a standard staple model in real-world customer retention scenarios, particularly in domains like telecommunications and finance, where interpretability is essential, and decisions such as the chosen threshold for classification must be explained to business stakeholders.

While my project work is not for clients or business-facing, LR still represents a strong modelling choice. It allows for meticulous evaluation, explainable behaviour, and meaningful benchmarks for comparison. LR’s parameters allow us to analyse how each feature contributes to the prediction, providing insight into model behaviour on our churn dataset. This interpretability was especially helpful in understanding how different configurations, such as regularisation strength or threshold values, influenced the model's performance.

6.2 Theory and Implementation

6.2.1 Theory Behind it

Logistic Regression is a supervised learning algorithm for binary classification. It works by calculating the probability that a given input belongs to the positive class (label = 1), which is a function of the input features. It computes a weighted linear combination of the encoded feature values in this case (tenure, monthly contract, payment method) and the bias term. The bias term shifts the logistic curve horizontally, allowing the Logistic Regression model to account for probability outcomes not centred around zero. This flexibility from the bias helps the model generalise unseen data well, where the probability of outcomes is not centred around the decision threshold.

A sigmoid function is applied to the raw logit scores calculated to map the logits into a probability between 0 and 1. A decision threshold is then applied to obtain a binary prediction from this probability: if the probability is at least 0.5, the instance is classified as class 1; otherwise, it is class 0. This approach allows the model to make discrete predictions. In my case, churn = 1 and non-churners = 0 and the confidence measure, is the probabilities for each prediction [14].

$$
\begin{aligned}
P(y=1) &= \sigma(\mathbf{w}\cdot\mathbf{x}+b) \\
&= \frac{1}{1+\exp\left(-(\mathbf{w}\cdot\mathbf{x}+b)\right)} \\
P(y=0) &= 1-\sigma(\mathbf{w}\cdot\mathbf{x}+b) \\
&= 1-\frac{1}{1+\exp\left(-(\mathbf{w}\cdot\mathbf{x}+b)\right)} \\
&= \frac{\exp\left(-(\mathbf{w}\cdot\mathbf{x}+b)\right)}{1+\exp\left(-(\mathbf{w}\cdot\mathbf{x}+b)\right)}
\end{aligned}
$$

Figure 12. Logistic regression probability equations for class 1 and class 0.

6.2.2 My Implementation

Like all models, my implementation of the Logistic Regression Classifier was done from scratch in Python and utilised Numpy. To encapsulate the theory learnt I broke down my LR model into the following functions:

Sigmoid Function:
This was a helper function and relatively straightforward to implement. However, to prevent instability in the sigmoid function, I correctly scaled the data in preprocessing, allowing for a more stable convergence in my training.

Model parameters: The model maintains a weight vector (one weight for each feature) and a single bias term. These were initialised to 0 at the start of training.

Fit: A fit method was implemented to train the model using batch gradient descent. On each epoch, the model computes predictions for all training samples in one vectorised operation (using matrix dot-products for efficiency). The difference between predicted probabilities and true labels is then used to calculate the gradients for each weight and the bias. My initial implementation consisted of no regularisation, but to prevent overfitting and improve generalisation, I incorporated L2 regularisation. The L2 regularisation adds a penalty proportional to the sum of the squared weights to the loss function. This discourages the model from assigning excessively large weights to any feature, thus regularising the learned parameters. In the gradient calculations, this resulted in an extra term of (λwj) added to the weight gradients, effectively pulling each weight towards zero on every update.

Predict: The prediction function classifies the unseen data as churned or non-churned. It calculates the sigmoid probability for each input (using the learned weights and bias) and then applies the 0.5 threshold to produce a binary outcome. This yields a list of predicted labels (0 or 1) for the input samples. The threshold was kept at the default 0.5 for this stage, with no adjustment , I will address the threshold tuning and precision-recall analysis carried out in later sections.

Loss computation: Loss computation is a key factor in Logistic Regression. It guides the model during training. I implemented binary cross-entropy loss, which compares the predicted probabilities to the actual labels, outputting a higher loss when the predictions are confident but wrong

```
loss = -np.mean(y * np.log(probabilities + 1e-10) + (1 - y) * np.log(1 -
probabilities + 1e-10)) # Add epsilon to prevent log(0)

l2_penalty = (self.lambda_reg / 2) * np.sum(self.weights**2)
return loss + l2_penalty # Add L2 regularisation term
```

I added some safety measures to the loss by adding a small epsilon value to the logarithmic terms. This ensured numerical stability by preventing any log(0) calculations from occurring and any edge cases where predicted probabilities are close to 0 or 1.

6.3 Threshold Tuning

Generally, logistic regression uses a default threshold of 0.5; however, given that my dataset is imbalanced, there are many more non-churners than churners. This imbalance leads to suboptimal performance caused by the model's inability to accurately identify churners (recall), which is a business's primary goal. As a result, accuracy alone can be misleading, but focusing on a balance of high recall and precision is often the goal in retaining and catching churners.

Threshold tuning was vital to addressing this problem. I decided to experiment with a range of threshold values from 0.1 - 0.9 in a **Jupyter Notebook**. This allowed me to empirically analyse what threshold would be suitable to balance the trade-off between recall (the model's sensitivity to actual churners) and precision (the proportion of correct churn predictions).

In my scenario, high recall was a priority, and ensuring precision didn't suffer too much was a second priority. To do this, I plotted a precision-recall curve.

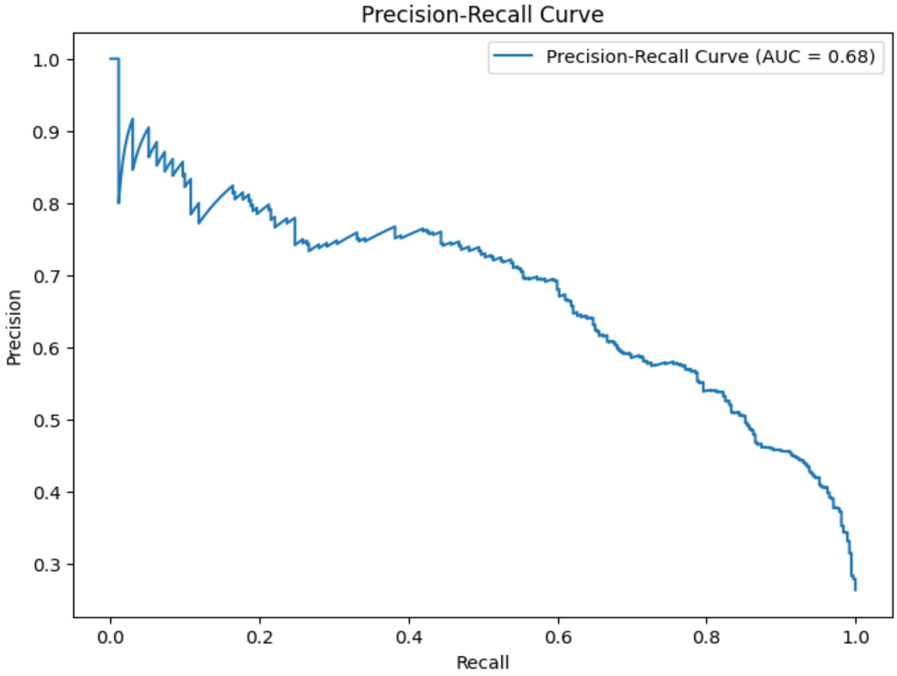


Figure 13. Precison-Recall graph for LR model

The PR plot demonstrates a clear trade-off in precision and recall. We can observe a clear downward trend in which, as the recall increases, the precision also decreases. The curve shows that capturing all possible churners requires a huge loss in precision, meaning that many churners would be flagged as churners. In a real-world environment, this would be very bad for the company as they would be wasting lots of resources, time and effort on customers who will never churn with targeted retention strategies.

The area under the curve (AUC-PR) is a metric that summarises the model's overall performance. For a perfect classifier, the AUC is equal to 1, meaning it classifies every data correctly. This metric is very suitable for my imbalanced dataset. Achieving an AUC score of 0.68, indicates a reasonable ability to distinguish between churners and non-churners. While this score is not ideal, it identifies room for improvement through techniques such as resampling and experimenting with different learning rates and epochs.

Utilising the PR plotting, experimentation, and parameter tuning, the optimal threshold is 0.49. This value offered the best trade-off between capturing a higher proportion of churners (recall) while maintaining acceptable precision. It also ensures that more churners were captured and a decrease in false positives, which is vital in customer retention strategies.

6.4 Weighted Logistic Regression

Threshold tuning does offer a solution to class imbalance. However, it is a post-training solution that does not change how the model learns but adjusts how the inputs are interpreted. In contrast, Weighted Logistic Regression considers class imbalance during training by asserting the greater importance of not misclassifying the minority class. This leads to WLR optimising its weights differently in training, causing the decision boundary to be altered as compared to a standard Logistic Regression model.

6.4.1 How does Weighted Logistic Regression Work?

A standard Logistic Regression treats all classes equally in training, which generally leads to a bias toward the majority class. Weighted logistics deals with this bias regression by assigning different weights to each class; usually, the minority class is given a higher weighting.

In training, the binary cross-entropy loss function is modified to incorporate these class weights explicitly. The class weights are generally a dictionary of label and weighting like {0: 1.0, 1: 2.0}, assigning a higher penalty to class 1. So, each sample's loss is scaled by the corresponding class weight. Thus, increasing the penalty for errors made on minority-class, in my scenario a higher penalty for misclassifying churners as non-churners.

Here is the formal equation for WLR including L2 regularisation:

$$L = -\frac{1}{N}\sum_{i=1}^{N} w_{y_i}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right] + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

Figure 14. WLR Loss function including l2 regularisation

where:

- w_{y_i} is the weight assigned to the class of the ith sample.
- $\widehat{y_i}$ is the predicted probability from the logistic sigmoid.
- λ represents the L2 regularisation term.

6.4.2 Implementation and adjustments.

When developing my **LR** model, I considered further possible development, such as a weighted version. Due to this, implementing weighted logistic regression became much simpler, and less redundant code was required. I structured my LR class in a modular and Object-oriented way, allowing my weighted version to inherit its functionalities and parameters. Through class inheritance, I override how the model trains (fit method), considering weights and penalties, and reused the core functionalities of prediction and sigmoid functions. This made my class functionalities readable and made testing easier and more flexible.

A challenging aspect was correctly integrating the class weights into the gradient update without affecting my L2 regularisation functionality. To solve this, I applied class weights only to the loss and error terms, not the L2 regularisation.

Below is a snippet of my overridden fit() function:

```
# WeightedLogisticRegression.fit()
sample_weights = np.array([self.class_weights[label] for label in y])

for _ in range(self.epochs):
    logits = np.dot(X, self.weights) + self.bias
    probabilities = self.sigmoid(logits)

    weighted_errors = sample_weights * (probabilities - y)

    dw = (1 / n_samples) * (
        np.dot(X.T, weighted_errors) + self.lambda_reg * self.weights
    )

    db = (1 / n_samples) * np.sum(weighted_errors)

    self.weights -= self.eta * dw
    self.bias -= self.eta * db
```

I also updated the loss function to reflect the use of class weights. As shown in the snippet below, each sample’s contribution to the loss was correctly scaled according to its class, while L2 regularisation was applied separately. This ensured that class weighting influenced training effectiveness without interfering with the regularisation’s role in controlling large weights.

```
loss = -np.mean(sample_weights * log_terms)
l2_penalty = (self.lambda_reg / 2) * np.sum(self.weights ** 2)
total_loss = loss + l2_penalty
```

6.5 WLR & LR: Evaluation and Results

6.5.1 Logistic Regression results:

The final model was configured with a learning rate of 0.001, 5000 training epochs, and no regularisation. As discussed before my chosen threshold value of **0.495** was selected.

Results are as follows:

- **Test Set**: Accuracy: 0.801 (80.1%), Precision: 0.604 (60.4%), Recall: 0.726 (72.6%), F1 Score: 0.659 (65.9%), F2 Score: 0.698 (69.8%)
- **Cross-Validation**: Accuracy: 0.781 (78.1%), Precision: 0.576 (57.6%), Recall: 0.667 (66.7%), F1 Score: 0.618 (61.8%), F2 Score: 0.646 (64.6%)

6.5.2 Weighted Logistic Regression Results:

The Weighted Logistic Regression model was configured with a learning rate of 0.001, 5000 training epochs, no regularisation and class weights of {0: 1.0, 1: 1.5}. My chosen threshold is same as the LR model (0.495).

Results are as follows:

- **Test Set**: Accuracy: 0.778 (77.8%), Precision: 0.558 (55.8%), Recall: 0.777 (77.7%), F1 Score: 0.649 (64.9%), F2 Score: 0.720 (72.0%)

- **Cross-Validation**: Accuracy: 0.767 (76.7%), Precision: 0.544 (54.4%), Recall: 0.743 (74.3%), F1 Score: 0.628 (62.8%), F2 Score: 0.692 (69.2%)

Before selecting my final configuration, an interesting experiment I carried out was inverse weightings with a lower threshold.

The configuration was {0: 1.0, 1: 0.5} with a lower threshold of 0.34. Here 0 is the non-churners class, and 1 is churners.

This meant that the minority class(churners) were down-weighted intentionally during training, making the model less sensitive to churners. However, the lower threshold essentially compensates for this by increasing the probability of churning occurring. My aim was to see whether training the model with a less sensitive approach to churners while using a more aggressive decision boundary leads to a better balance between precision and recall. The results from this were an increase in recall 81.2% and a precision of 52.8%. While the recall was impressive, it came at a high cost to precision, making it less practical in a business context (too many false positives), which led me not to use this configuration. Despite this, it reinforced the idea that adjusting class weights and decision thresholds can be a powerful way to fine-tune model performance rather than relying solely on algorithmic optimisation.

6.5.3 A head-to-head of WLR VS LR

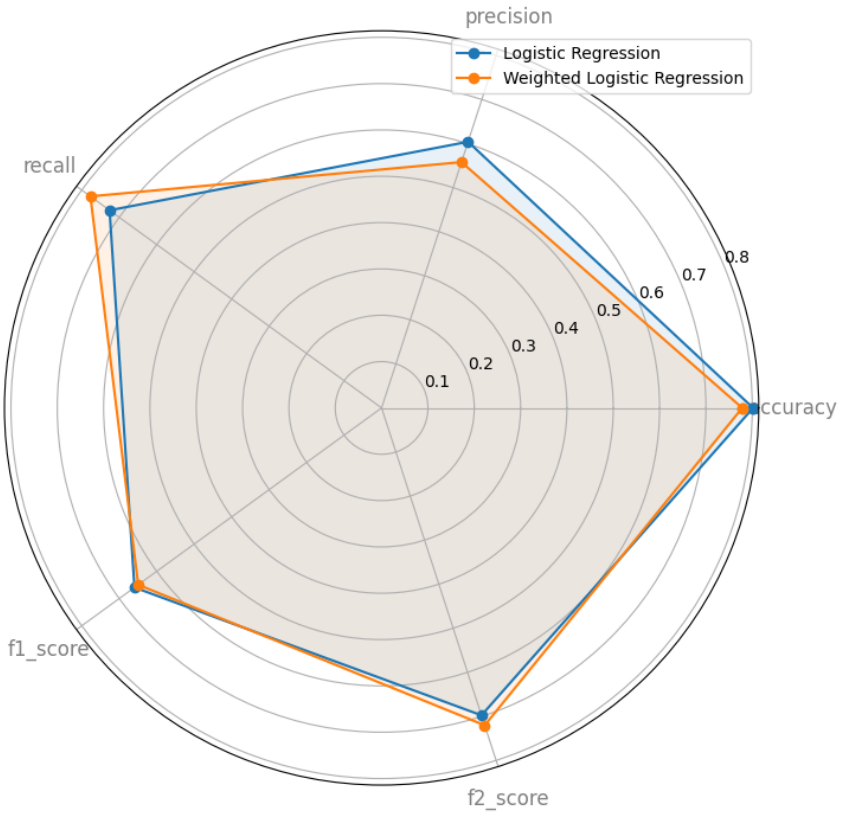


Figure 15. Radar plot (WLR vs LR)

The radar chart in Figure 15 visually compares both models across all evaluation metrics. We can observe from the Figure that Weighted Logistic regression outperforms standard Logistic Regression in **F2** and **recall,** the most important metric in maximising churn detection. However, this comes at the cost of accuracy and precision, which is understandable as there will always be a trade-off. This shows that WLR provided a more recall-oriented model and is suitable for businesses where ensuring that all potential churners are detected is crucial. Even if is at the risk of more false positives.

Chapter 7: **Advanced Models**

This chapter will focus on two advanced classification algorithms, Random Forest (RF) and Extreme Learning Machine (ELM). Random Forest is an ensemble method built on decision trees and is designed to improve generalisation and reduce overfitting. ELM is a neural-network-based approach that forgoes iterative training for an extremely fast learning process. By applying both models to my churn prediction problem, I aim to evaluate whether their architectural differences (ensemble and Neural Net) compared to my previous models translate into meaningful performance gains.

7.1 Extreme Learning Machines (ELM)

7.1.1 Background and Theory

Extreme Learning Machine (ELM) is a relatively recent approach to training single-hidden-layer feedforward neural networks. Proposed by Huang et al. (2006), ELM's key innovation is that it eliminates the iterative weight update process (no backpropagation is needed). Instead, the hidden layer's weights are assigned randomly and never adjusted, and only the output weights are learned. This is computed in a single step using a direct algebraic solution known as the Moore-Penrose pseudoinverse. [15] [16]

Despite this unconventional strategy, ELM theory shows that it can approximate complex decision boundaries if the hidden layer is sufficiently large and uses a non-linear activation. In other words, the random hidden neurons act as a diverse set of non-linear features, and a simple linear model is fitted on top of them to solve the classification. This results in extremely fast training; hence, the title "Extreme Learning" is given while still achieving good generalisation in many cases [15].

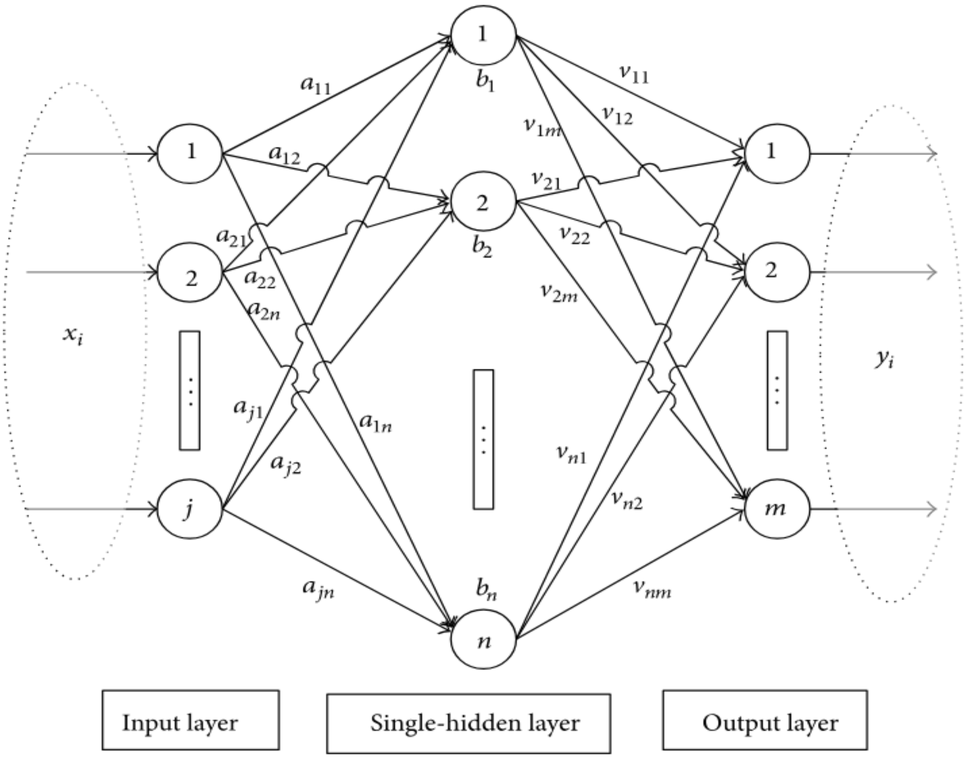


Figure 16. Single Hidden Layer Feedforward Neural Network (SLFN) used in Extreme Learning Machines. Image credits: Shifei Ding

While researching, the author claims ELM tends to provide excellent generalisation performance at an extremely high learning speed, giving me further reason to develop an ELM model. The primary motivation for using ELM in the churn project was to experiment with a neural network-like model that could potentially handle complex feature interactions but with a much simpler training process than traditional neural networks. Importantly, ELM can help sidestep the often time-consuming task of tuning an optimiser or waiting for many epochs of training given that the random weights are set. Since only the output weights are learned, the model solves for them in a single step via a linear system, making training very fast. However, we must carefully choose the number of hidden nodes and activation function, as these become the critical hyperparameters for ELM's performance. It is important since we do not adjust weights iteratively but rather rely on the hidden layer's random mapping to capture the underlying patterns in the telecom data.

7.1.2 My Implementation and Issues in development

I developed an ELM class in which different activation functions, number of nodes and random state are the parameters. For this class I have allowed for 3 different types of activation functions to be selected (Sigmoid, Relu, tanh). The random state parameter is for reproducibility, since ELM relies on random initialisation.

Core Implementation features:

- **Fit() :** A very Straightforward method, I set the random seed and randomly generate the input weights and biases for the hidden layer. In my code the hidden weights is a matrix shape consisting of hidden nodes and number of features and these contained the random values generated. Next, I compute the hidden layer output **matrix H**. This involves taking the dot product of the input data X with the transpose of input weights, adding the bias for each hidden node, and then applying the chosen activation function.

  During my implementation I ran into matrix shape issues caused by dot products between input data and the weight matrices. It was very tedious as it wasn’t very clear the exact causes of this issue. To resolve this, I had to revisit the **ELM** and carefully trace through the matrix operations and transposing matrices where necessary.

- **Activation():** The purpose of the activation function is to model the nonlinear relationships in our data. Without it our Neural network would behave like a linear model even if we have many layers ,defeating the purpose of a neural network’s ability to capture patterns. I have implemented support for Sigmoid, Tanh, and ReLU for the purpose of further investigations which I will discuss next.

- **Hidden_layer_output():** This function applies the hidden layer transformation essentially projecting the input data into a higher-dimensional non-linear space. This hidden layer is critical for enabling the output layer to learn about our churn dataset effectively.

  Below is a code snippet for this function. It computes a dot product between the input weights and the transposed input data. It then , adds the bias for each hidden node, and applies the chosen activation function.

```
X = X.T # Transpose for matrix multiplication

G = np.dot(self.input_weights, X) + self.biases # pre activation

return self.activation_function(G)
```

Code snippet for hidden layer output function ()

- **Output weight calculation:** After I had computed the hidden layer output. I utilised Moore-Penrose pseudoinverse to solve for the output weights in a single step.

  ```
  self.output_weights = np.dot(np.linalg.pinv(H_T), y.T)
  ```

  This line of code is crucial to ELM as it removes the need for iterative learning and computes using least squares. This step was bit difficult due to ensuring the transposes are correct and utilising the right NumPy function to emulate Moore-Penrose pseudoinverse. I had initially considered writing my own pseudoinverse function before discovering that numpy.linalg.pinv already handled this efficiently and reliably.

7.1.3 Impact of Hidden Neuron Count on ELM Performance.

The number of hidden neurons in the ELM model is a crucial hyperparameter. It controls the model's ability to capture complex and nonlinear patterns in our churn data. Since **ELM** is a single-feed neural network, the input weights are fixed and random. This makes the hidden neurons responsible for projecting the data into a higher-dimensional space. Too few neurons can lead to underfitting, while too many may introduce redundancy or instability. This is why I have carried out an experiment to select the correct number of hidden nodes to balance the model complexity and generalisation.

For this experiment, I have selected **F2 score** as the main metric as it prioritises **recall** more than precision. I used increments of 50 neurons, ranging from 50 to 650, and tested performance across three activation functions: Sigmoid, Tanh, and ReLU. Below is a figure showing the results of my experimentation.

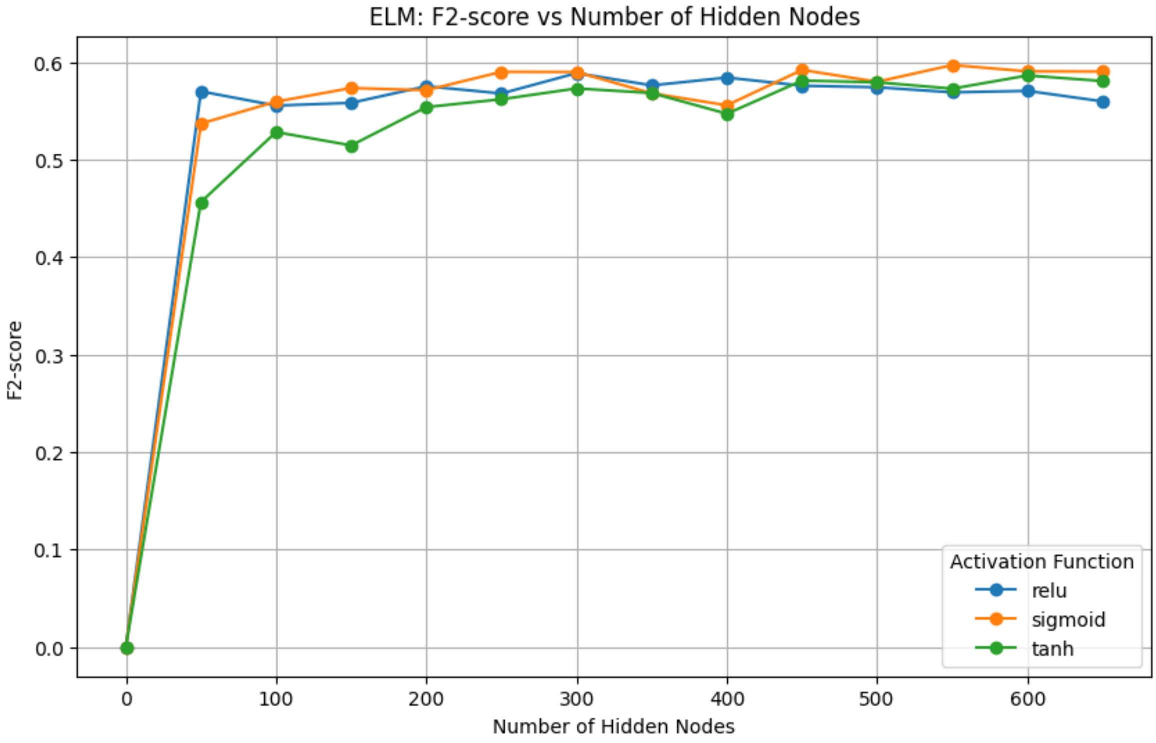


Figure 17. F2-score vs. Number of Hidden Neurons for different activation functions (Sigmoid, ReLU, Tanh).

From the plotting, we can observe that there is a rapid performance increase across all activation functions until the 100 neurons mark. This means the model is learning, and toward the 300-neuron mark, all activation functions begin to stabilise. Beyond 300, there were many fluctuations, and Relu showed diminishing returns. Based on these observations, I prioritised a balance between stability and performance. Therefore, I selected **400 neurons for sigmoid**, **300 for ReLU**, and **300 for tanh** as my final hidden node configurations for GUI integration and evaluation.

7.1.4 ELM Final Results and Evaluation

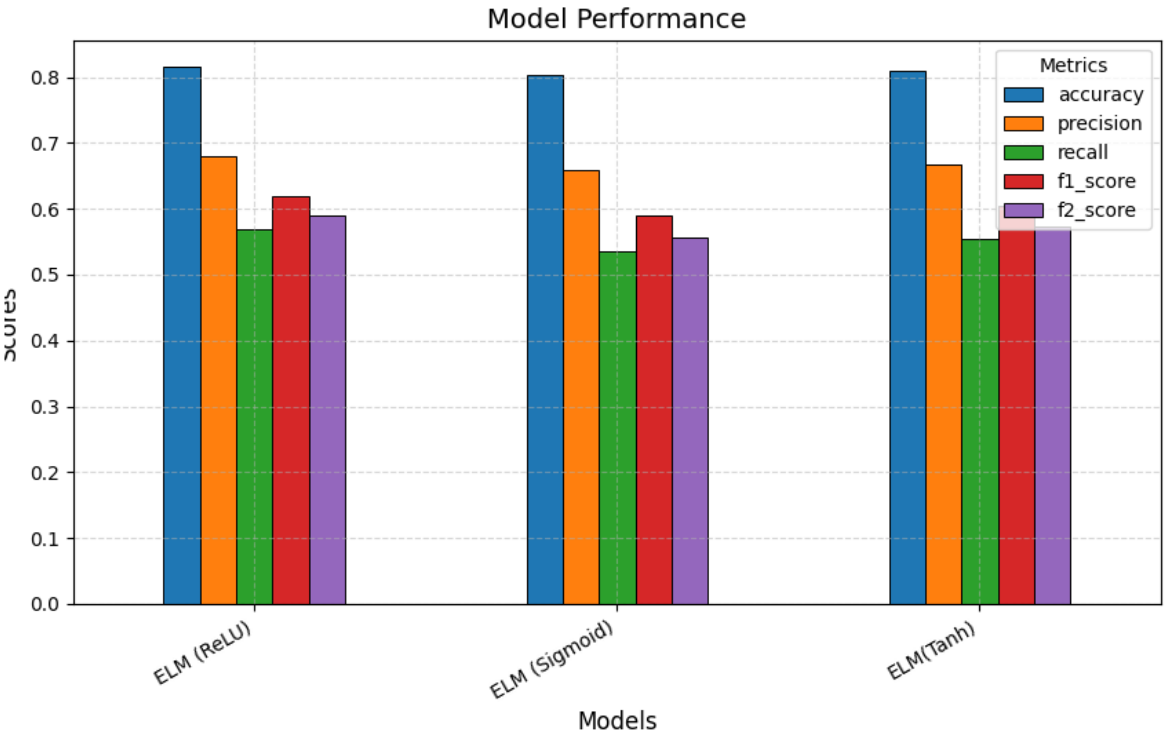


Figure 18. Test set performance of ELM models using different activation functions.

Test Set Results:

- **ReLU**: Accuracy: 0.815, Precision: 0.680, Recall: 0.570, F1: 0.620, F2: 0.589
- **Sigmoid**: Accuracy: 0.804, Precision: 0.659, Recall: 0.535, F1: 0.591, F2: 0.556
- **Tanh**: Accuracy: 0.809, Precision: 0.667, Recall: 0.554, F1: 0.605, F2: 0.573

Cross-Validation Results:

- **ReLU**: Accuracy: 0.787, Precision: 0.629, Recall: 0.487, F1: 0.549, F2: 0.510
- **Sigmoid**: Accuracy: 0.782, Precision: 0.611, Recall: 0.496, F1: 0.547, F2: 0.515
- **Tanh**: Accuracy: 0.777, Precision: 0.600, Recall: 0.479, F1: 0.533, F2: 0.4w99

The results show that **ELM with ReLU** activation provided the best overall performance on the test set. It achieves the highest accuracy and F1/F2 scores while also generalising well in 5-fold cross-validation. Although ELM (RELU) performed best, it is not suitable for an actual company to use this model to make decisions. This is mainly due to its lack of interpretability and sensitivity to random initialisation, which makes it difficult to justify in decision-making environments where transparency is essential.

7.2 Random Forest

7.2.1 Background Theory

Ensemble learning is a machine learning technique in which multiple base models are trained and aggregated to produce a more robust and accurate model. **Random Forest** is an ensemble learning method that constructs a collection of decision trees as the base learners. It is a very well-known model, as it does not rely on a single tree, which can be prone to overfitting. The idea here is that by combining many trees, each trained on a different random subset of the data, the model can average out the errors of individual trees. This is known as Bagging or Bootstrap Aggregating, where each tree is trained on a bootstrap sample (sampling training data with replacement) [11]**.**

Bagging introduces diversity among the trees, reducing the likelihood that the ensemble will overfit to specific patterns or noisy data. Beyond sampling the data, Random Forest also introduces randomness during tree-building by considering only a random subset of features at each split. This random feature selection prevents dominant features from appearing in every tree, which enhances model diversity and decreases the correlation between individual trees.

Relating this to my churn prediction problem, a dominant feature such as contract type could heavily influence all trees if feature selection was not randomised.
However, by limiting each tree to consider only a subset of features at each decision node, Random Forest ensures that other relevant attributes (such as tenure, monthly charges, or payment method) also influence the ensemble's decisions. This then leads to an improved accuracy and better generalisation on unseen data.

After training, each tree in the forest votes on the predicted class, and the final output is determined by majority voting. This voting mechanism helps smooth out any individual errors, enhancing the overall reliability of the predictions.

7.2.2 My implementation

To implement my Random Forest class, I utilised my previously developed DecisionTree class.

New key parameters introduced were:

- **n_estimators**: the number of trees to train in the forest.
- **max_depth**: the maximum depth each individual tree can grow to.
- **min_samples_split**: Minimum samples required to split a node.
- **n_features** (optional): Number of features to consider when looking for the best split.

A challenging aspect was ensuring efficiency in my implementation. Initially, in my early versions of the Random Forest it took significantly longer to train compared to sci-kit-lean's implementation, which completed training in a fraction of the time. This prompted me to investigate where the bottleneck was occurring. I realised that I was training each tree sequentially, one after the other, like a queue, naturally leading to a slower runtime. To resolve this issue, I implemented parallelism in training multiple trees using **joblib**. This drastically reduced the training time and allowed for testing a wider range of values for the number of trees.

A key aspect of random forest was the usage of bootstrapped sample for each tree. To achieve this, I created a helper function that randomly sampled the training data with replacement, producing a dataset of the same size as the original. This essentially ensured that each tree utilises a random sample from the training set thus introducing the variance needed in ensemble models to generalise better.

7.2.3 Results and Evaluation

After validating the implementation with unit tests and comparing it to Scikit Learn, I applied my Random Forest model to the customer churn dataset. I trained an ensemble of 100 trees (each with a max depth of 10, using Gini impurity), as this configuration was what I found to be best in my hyper parameter investigation.

Metric	*Test Set*	*5-fold Cross-validation*
Accuracy	*0.8004*	*0.7913*
Precision	*0.654*	*0.6314*
Recall	*0.543*	*0.5175*
F1 Score	*0.5898*	*0.5685*
F2-Score	*0.608*	*0.5367*

Figure 19. Random forest table of Results

As expected, the random forest generalises well to unseen data; this is evident from the results table as scores differ slightly between the cross-validation and the test set. The random forest model has an impressive accuracy of 80% in the test set predictions. However, this is overshadowed by the model's relatively modest recall of 54.3%. This means that the model can only predict at most 54.3% of churners, which can be financially damaging in a business context. The F1 and F2 scores further reflect this trade-off, indicating a less robust performance. Also, the higher precision score of 64.5% is acceptable, given that non-churners are the majority class, and suggests that the RF model is more conservative with its predictions.

To conclude, these results are understandable and lead back to the underlying class imbalance issue. There is simply not enough data for the model to learn patterns on customer churners. As such, further techniques may be required, such as resampling the data, which I will discuss in the upcoming chapters.

Chapter 8: **Handling Class Imbalance**

8.1 Motivation

Class imbalance is a prevalent issue, especially in my churn prediction project**.** Non-churners outnumber churners significantly, causing the models to favour the majority class. Since the primary focus is on identifying churners, this often leads to poor recall and a false sense of security from seemingly high accuracy.

While threshold and hyperparameter tuning I carried out help mitigate these issues, they operate during or after model training. This means they do not change the underlying sample distribution, which limits the model's ability to learn meaningful patterns in the minority class.

To address this, I explored two resampling techniques that directly alter the dataset before training:

- **SMOTE (Synthetic Minority Oversampling Technique):** Generates synthetic samples to increase the number of minority class examples.

- **Tomek Links:** Removes overlapping samples between classes to clean ambiguous decision boundaries.

The aim is to evaluate whether resampling improves my existing model performances, especially in **recall and F2-score**.

8.2 SMOTE (Synthetic Minority Oversampling Technique)

8.2.1 What is SMOTE?

Synthetic Minority Over Sampling is a resampling technique that creates new synthetic samples for the minority class. Duplicating existing minority samples can help balance the dataset; however, it does not provide new information for the model to learn from. In contrast, **SMOTE** is more effective as it generates **new examples** by interpolating between a given minority instance and one of its *k*-nearest minority neighbours.

To create new synthetic samples, a sample from the minority class is randomly selected, and a random neighbour from the minority class is then selected using KNN. The difference between the two samples is then calculated and multiplied by an interpolation value between 0 and 1. This results in a new point that lies along the line segment between the original sample and its neighbour, which is known as a synthetic sample.
This synthetic sample introduces a variation that allows the model to learn generalisable patterns while reducing the risk of overfitting to repeated or duplicated instances. The SMOTE techniques were originally proposed by Chawla et al. for addressing class imbalance and have utilised her algorithm in aid of developing my own implementation [18].

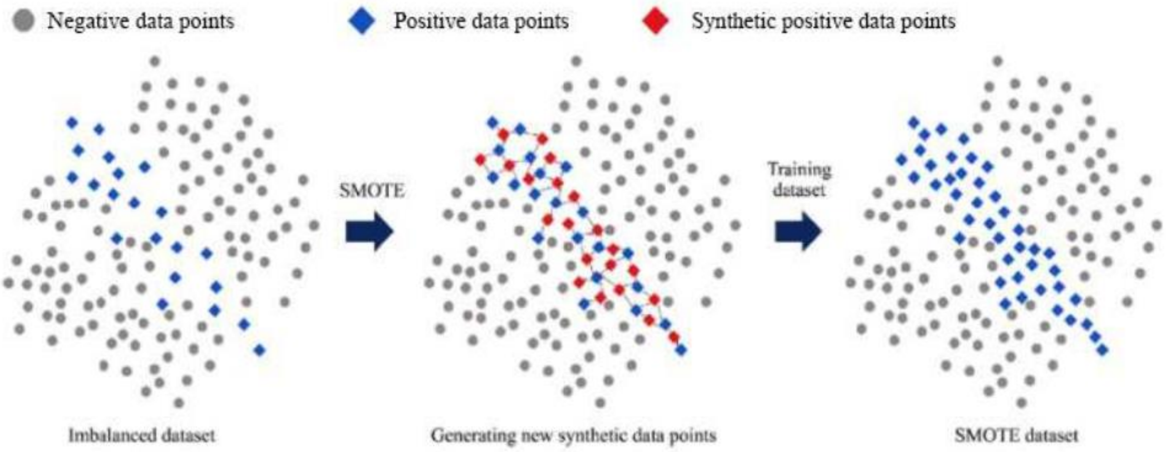


Figure 20. Synthetic Minority Oversampling Technique (SMOTE) Algorithm [17]

8.2.2 Applying SMOTE to churn dataset and handling data Leakage

Utilising my already developed KNN class and the SMOTE algo proposed by Chawla et al, I implemented the SMOTE resampling script. My implementation can be found in the **preprocessing folder**. I have utilised a k-value of 5 for my KNN and also compared to Imblearn library implementation of SMOTE as a form of regression testing to ensure that my implementation was valid.

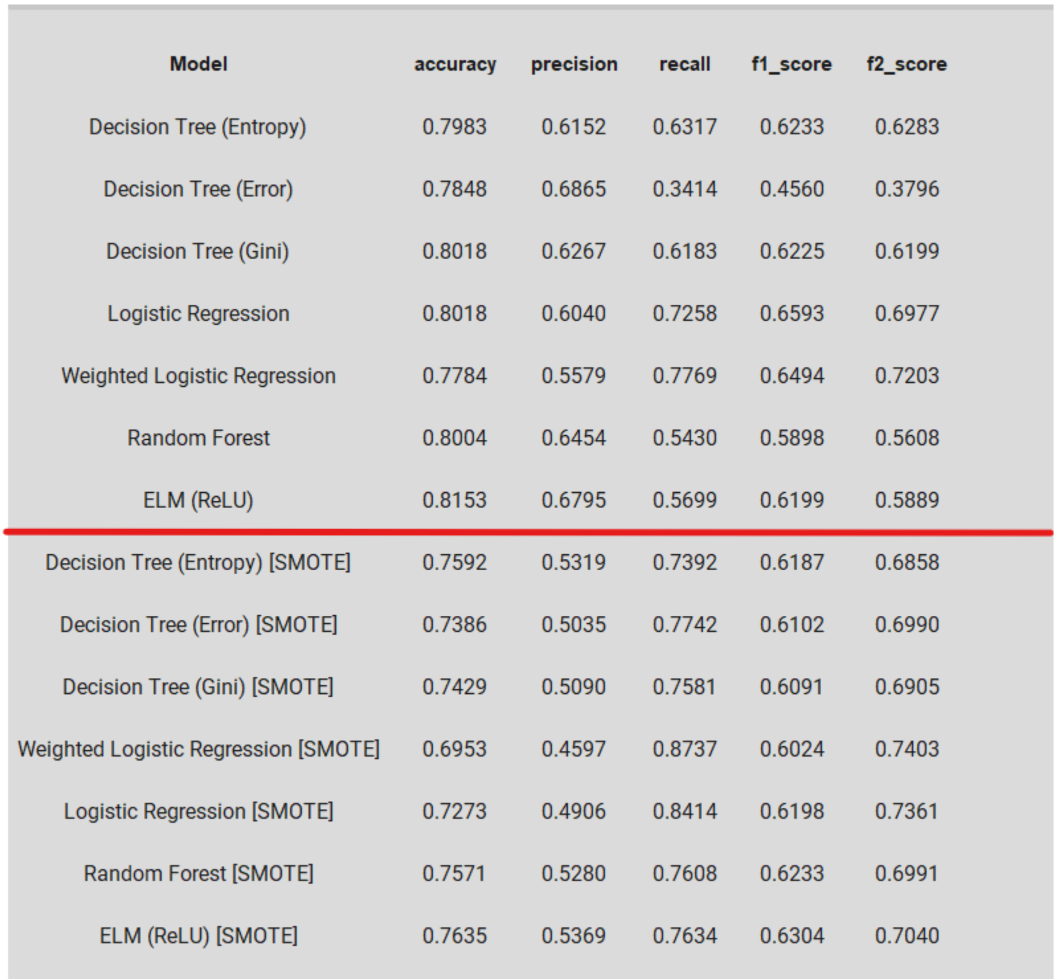
A key issue I came across in my research of SMOTE is when to apply SMOTE in the pipeline. Initially, I implemented SMOTE on the whole dataset before carrying out the train-test split. My results showed immediate boosts to all metrics, reaching an f1 score in the high 80%. However, I quickly came to realise my results were overly optimistic due **to data leakage**. The leakage occurs because SMOTE generates synthetic samples based on existing data points. If applied before splitting, some of the synthetic samples based on the training data can closely resemble samples in the test set. This would indirectly expose the model to test data during training, leading to evaluation results that are exceptional but are actually misleading.

To mitigate this issue, I split the churn data into train and test sets and applied the SMOTE preprocessing script only to the training set. This approach ensured that ML models do not train on synthetic data derived from our test set. When running my models on the SMOTE dataset, my performance scores dropped, and this was to be expected. Nevertheless, my results are valid now as there is no data leakage.

8.2.3 Observations and Results

To evaluate SMOTE resamples impact I have created a separate SMOTE dataset and ran my models with same parameter configurations on the SMOTE resampled dataset. Using same hyper parameters ensured a fair comparison as performance changes are due to resampling and not tuning.

Below is my results table including SMOTE and non-SMOTE resampling



Model	accuracy	precision	recall	f1_score	f2_score
Decision Tree (Entropy)	0.7983	0.6152	0.6317	0.6233	0.6283
Decision Tree (Error)	0.7848	0.6865	0.3414	0.4560	0.3796
Decision Tree (Gini)	0.8018	0.6267	0.6183	0.6225	0.6199
Logistic Regression	0.8018	0.6040	0.7258	0.6593	0.6977
Weighted Logistic Regression	0.7784	0.5579	0.7769	0.6494	0.7203
Random Forest	0.8004	0.6454	0.5430	0.5898	0.5608
ELM (ReLU)	0.8153	0.6795	0.5699	0.6199	0.5889
Decision Tree (Entropy) [SMOTE]	0.7592	0.5319	0.7392	0.6187	0.6858
Decision Tree (Error) [SMOTE]	0.7386	0.5035	0.7742	0.6102	0.6990
Decision Tree (Gini) [SMOTE]	0.7429	0.5090	0.7581	0.6091	0.6905
Weighted Logistic Regression [SMOTE]	0.6953	0.4597	0.8737	0.6024	0.7403
Logistic Regression [SMOTE]	0.7273	0.4906	0.8414	0.6198	0.7361
Random Forest [SMOTE]	0.7571	0.5280	0.7608	0.6233	0.6991
ELM (ReLU) [SMOTE]	0.7635	0.5369	0.7634	0.6304	0.7040

Figure 21. Performance metrics of all models trained on the SMOTE-resampled dataset

As observed from the table of results above there is a clear increase in **Recall** across all models for **SMOTE** resampled dataset. For example, Logistic Regression's recall jumped from 0.7258 to **0.8414**, and Weighted Logistic Regression from 0.7769 to **0.8737**. This is to be expected as there is more data on the minority class (churners) for the model to learn from. However, this improvement came at the cost of precision in almost all models for instance Logistic Regression's precision fell from 0.6040 to **0.4906**, and accuracy from 0.8018 to **0.7273**. While this can be possibly fixed by tuning it reflects the main theme being trade-off between recall-precision. The models become more sensitive to churners however it also misclassifies more churners and depending on business goals of a company such as taking the risks of misclassifying a higher number of churners to determine more possible churners than SMOTE can be utilised.

8.3 TOMEK Links Resampling

8.3.1 What is Tomek – Links Resampling?

We have looked at SMOTE an oversampling technique I will now look at a different resampling technique, Tomek Links. Tomek Links is a data-level method in which we under sample the majority class. It works by removing samples in the majority class that overlap and create some confusion around the decision boundary.

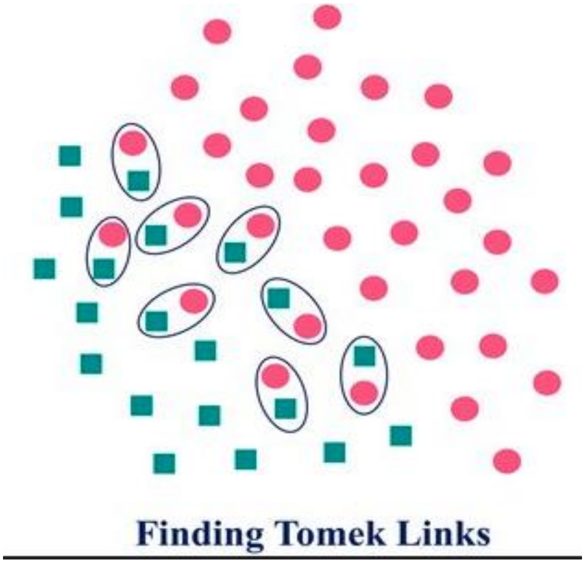


Figure 22. Illustration of Tomek Links — majority (red) and minority (blue) class samples with overlapping pairs highlighted. Image credits: [20]

These borderline samples are known as **Tomek Links**. For a pair of instances, i.e. samples, to form a Tomek Link, they must satisfy two conditions:

1. The two samples must belong to different classes

2. They are each other's closest neighbours

If and only if these two conditions are met, the pair is considered a Tomek Link. and the major class in the Tomek link will be removed. The reason why the majority class sample is removed from the Tomek link pair is that the minority class sample carries valuable information about rare patterns the model needs to learn. Removing majority class samples, which are more abundant, helps clean the decision boundary without sacrificing critical minority class knowledge.

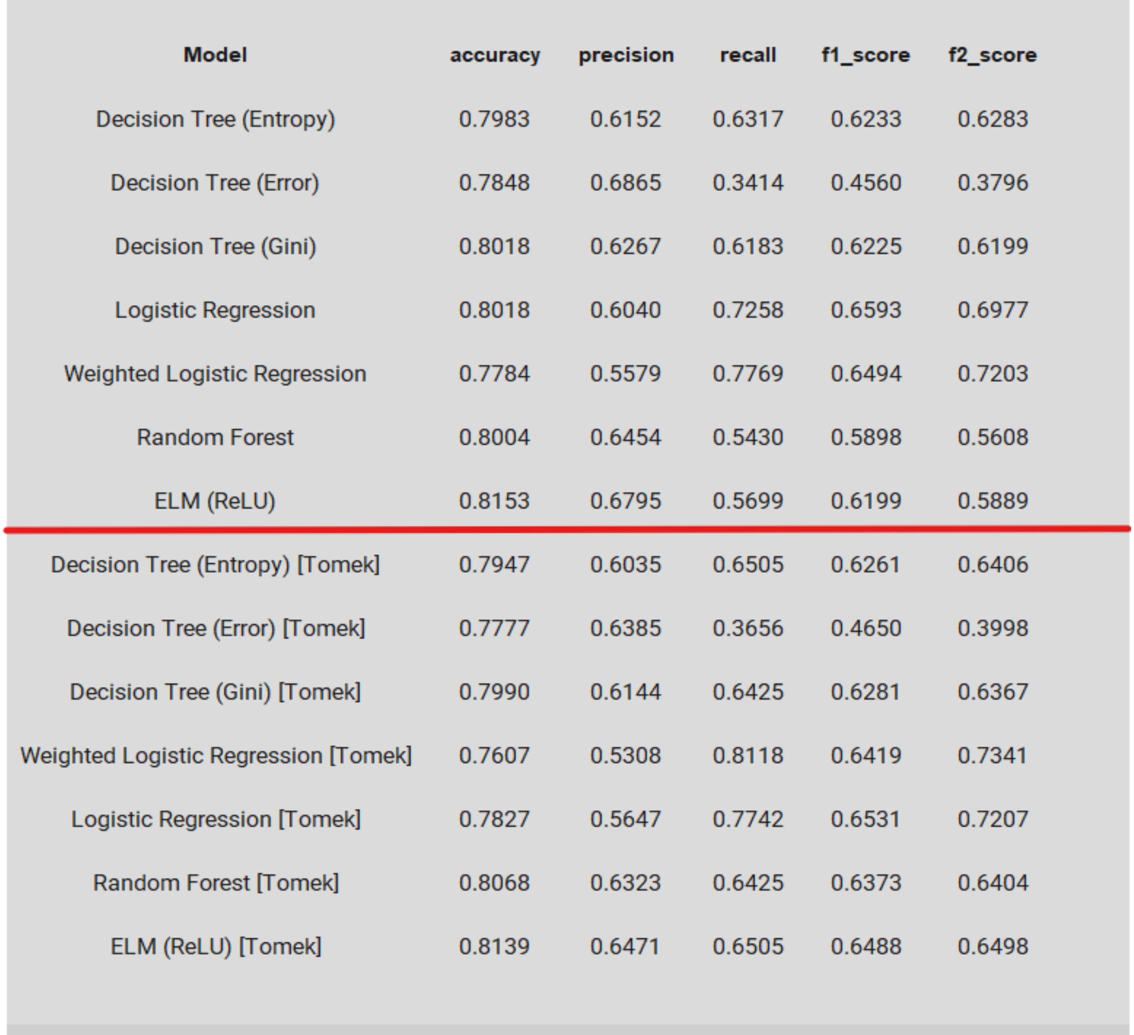
This under sampling technique was introduced by Ivan Tomek and later discussed by Batista et al. [19]. I have applied this theory to resample the churn dataset I'm working on.

8.3.2 Applying Tomek Links to churn dataset

To evaluate the effects of Tomek Links, I implemented a custom resampling script, which can be found in my preprocessing folder alongside the SMOTE script. My implementation uses a Euclidean distance class to identify Tomek Link pairs within the dataset. To ensure correctness, I also performed regression testing by comparing my output with the Imblearn library’s Tomek Links implementation.

Although I originally planned to reuse my existing KNN class for finding nearest neighbours, I encountered a design issue in my KNN implementation. My knn implementation was not set up to return neighbour **indices**, which are essential for identifying Tomek Links. Refactoring the KNN class to return indices would have impacted other classes and scripts, such as my SMOTE class. To avoid causing a chain reaction of issues I created dedicated helper functions, get_closest_neighbour_index() and euclidean_distance().These functions focus on identifying the mutual nearest neighbour relationships between samples using Euclidean distance a necessity for finding Tomek links.

8.3.3 Results after applying Tomek-Link Resampling



Model	accuracy	precision	recall	f1_score	f2_score
Decision Tree (Entropy)	0.7983	0.6152	0.6317	0.6233	0.6283
Decision Tree (Error)	0.7848	0.6865	0.3414	0.4560	0.3796
Decision Tree (Gini)	0.8018	0.6267	0.6183	0.6225	0.6199
Logistic Regression	0.8018	0.6040	0.7258	0.6593	0.6977
Weighted Logistic Regression	0.7784	0.5579	0.7769	0.6494	0.7203
Random Forest	0.8004	0.6454	0.5430	0.5898	0.5608
ELM (ReLU)	0.8153	0.6795	0.5699	0.6199	0.5889
Decision Tree (Entropy) [Tomek]	0.7947	0.6035	0.6505	0.6261	0.6406
Decision Tree (Error) [Tomek]	0.7777	0.6385	0.3656	0.4650	0.3998
Decision Tree (Gini) [Tomek]	0.7990	0.6144	0.6425	0.6281	0.6367
Weighted Logistic Regression [Tomek]	0.7607	0.5308	0.8118	0.6419	0.7341
Logistic Regression [Tomek]	0.7827	0.5647	0.7742	0.6531	0.7207
Random Forest [Tomek]	0.8068	0.6323	0.6425	0.6373	0.6404
ELM (ReLU) [Tomek]	0.8139	0.6471	0.6505	0.6488	0.6498

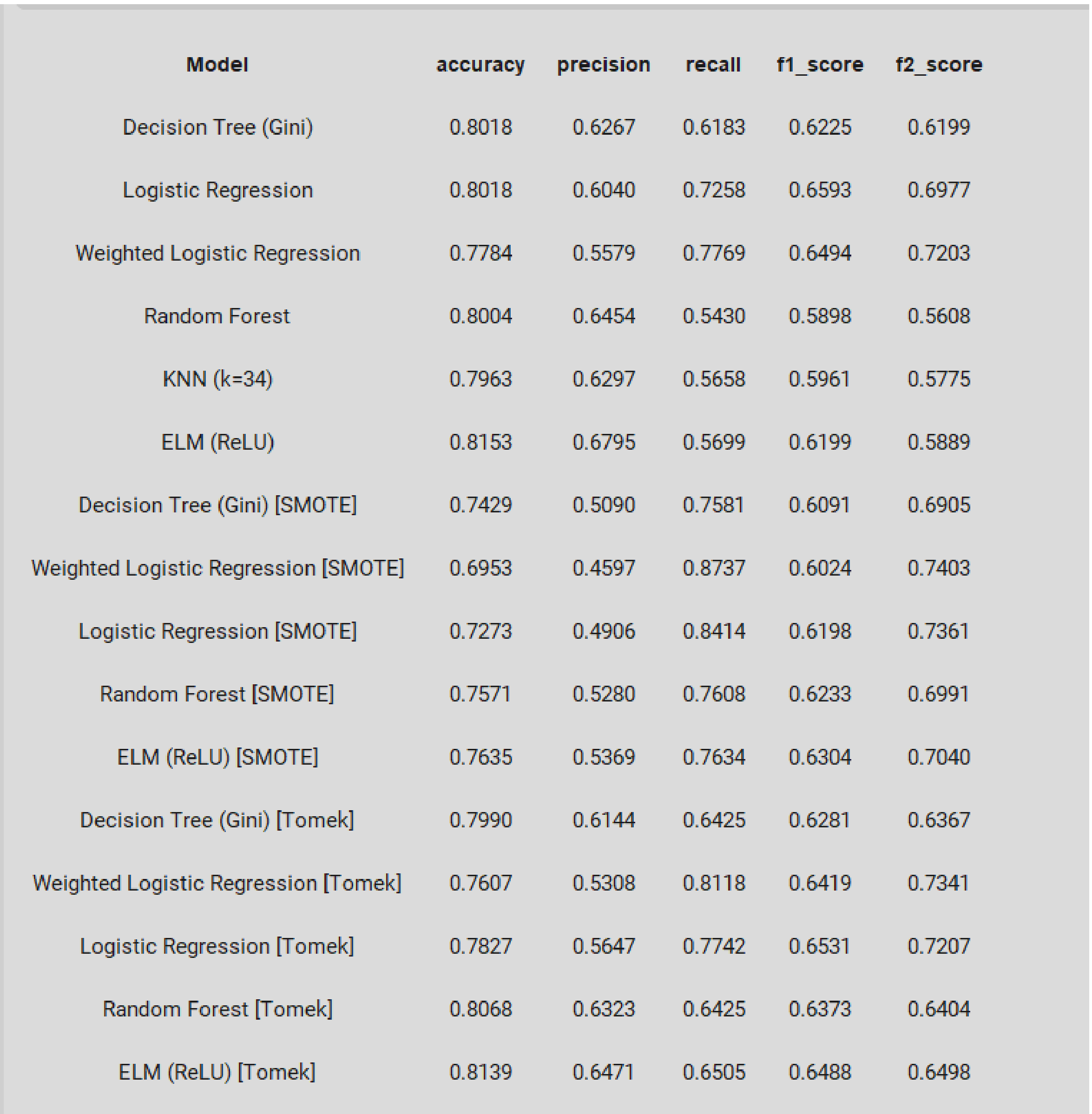
Figure 23. Table of Results comparing TOMEK vs Original

My results show a mixed impact of Tomek resampling. Some models, such as Logistic Regression's recall, improved from **0.7258 to 0.7742**, and Weighted Logistic Regression improved from **0.7769 to 0.8118**. This was to be expected, considering the goal of resampling is to clean the decision boundaries. However, similar to SMOTE, there was a clear trend in which precision decreased slightly. This is understandable, given we are not introducing new data but removing data from the majority class(non-churners). It can sometimes improve generalisation but may also lead to reduced accuracy due to fewer training examples. For instance, the recall of the Decision Tree (Error) improved, but its F1 and F2 scores remained low, likely due to a low number of correctly predicted positives overall.

An interesting observation I noted is that ELM showed a slight improvement across all metrics. By removing the noisy borderline samples through Tomek filtering, the hidden layer mapping became more stable, allowing for the output layer to generalise effectively.

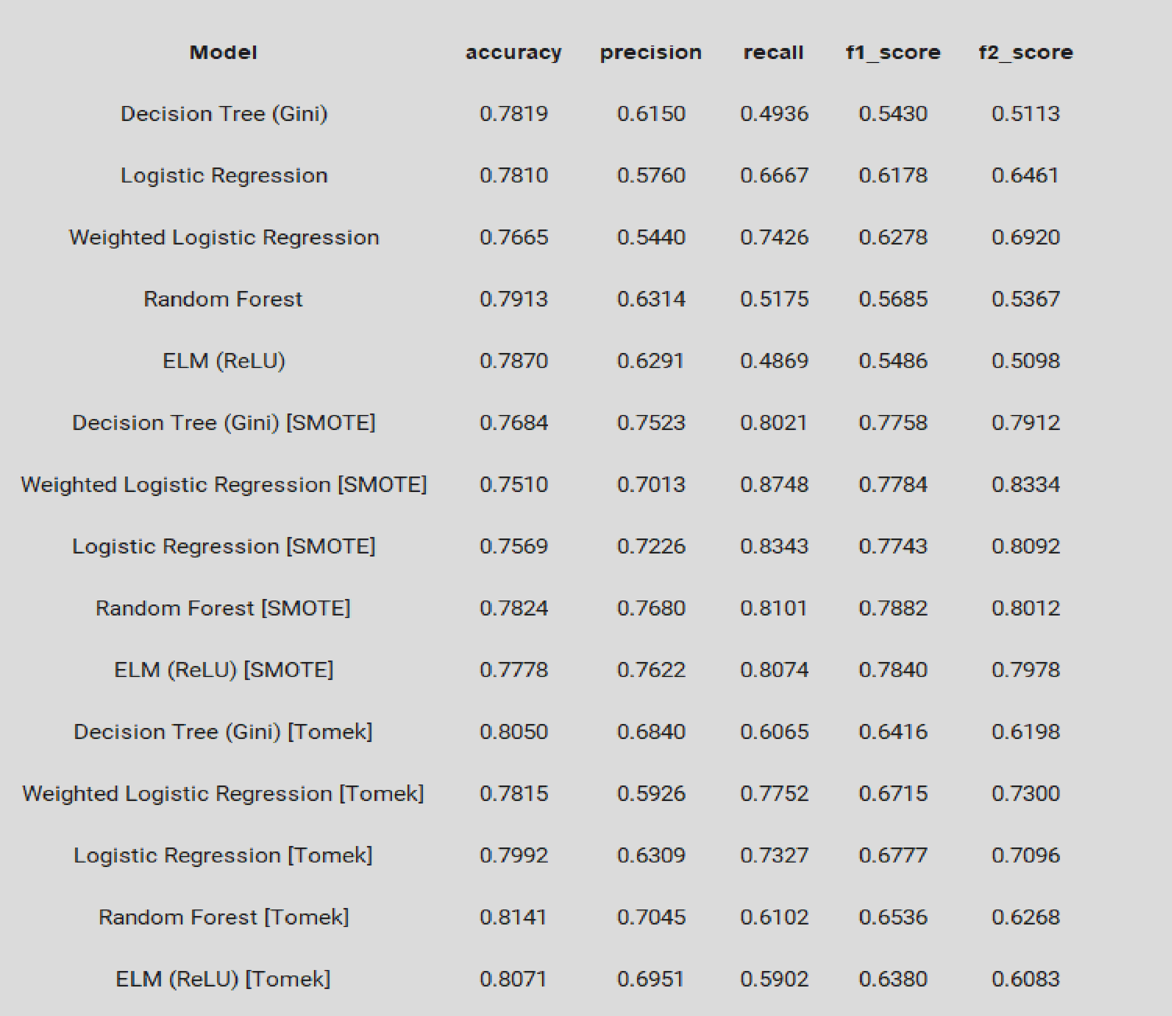
Chapter 9: **Overall Performance Evaluation**

In my previous chapters, I have evaluated each model and its results individually. In this chapter, I will evaluate all models comparatively, including the model trained with resampled data, to decide which model is best for churn predictions. I will utilise the performance scores and account for the model in a business context to carry out my observations.



Model	accuracy	precision	recall	f1_score	f2_score
Decision Tree (Gini)	0.8018	0.6267	0.6183	0.6225	0.6199
Logistic Regression	0.8018	0.6040	0.7258	0.6593	0.6977
Weighted Logistic Regression	0.7784	0.5579	0.7769	0.6494	0.7203
Random Forest	0.8004	0.6454	0.5430	0.5898	0.5608
KNN (k=34)	0.7963	0.6297	0.5658	0.5961	0.5775
ELM (ReLU)	0.8153	0.6795	0.5699	0.6199	0.5889
Decision Tree (Gini) [SMOTE]	0.7429	0.5090	0.7581	0.6091	0.6905
Weighted Logistic Regression [SMOTE]	0.6953	0.4597	0.8737	0.6024	0.7403
Logistic Regression [SMOTE]	0.7273	0.4906	0.8414	0.6198	0.7361
Random Forest [SMOTE]	0.7571	0.5280	0.7608	0.6233	0.6991
ELM (ReLU) [SMOTE]	0.7635	0.5369	0.7634	0.6304	0.7040
Decision Tree (Gini) [Tomek]	0.7990	0.6144	0.6425	0.6281	0.6367
Weighted Logistic Regression [Tomek]	0.7607	0.5308	0.8118	0.6419	0.7341
Logistic Regression [Tomek]	0.7827	0.5647	0.7742	0.6531	0.7207
Random Forest [Tomek]	0.8068	0.6323	0.6425	0.6373	0.6404
ELM (ReLU) [Tomek]	0.8139	0.6471	0.6505	0.6488	0.6498

Figure 24. Test set performance comparison across all selected model variants



Model	accuracy	precision	recall	f1_score	f2_score
Decision Tree (Gini)	0.7819	0.6150	0.4936	0.5430	0.5113
Logistic Regression	0.7810	0.5760	0.6667	0.6178	0.6461
Weighted Logistic Regression	0.7665	0.5440	0.7426	0.6278	0.6920
Random Forest	0.7913	0.6314	0.5175	0.5685	0.5367
ELM (ReLU)	0.7870	0.6291	0.4869	0.5486	0.5098
Decision Tree (Gini) [SMOTE]	0.7684	0.7523	0.8021	0.7758	0.7912
Weighted Logistic Regression [SMOTE]	0.7510	0.7013	0.8748	0.7784	0.8334
Logistic Regression [SMOTE]	0.7569	0.7226	0.8343	0.7743	0.8092
Random Forest [SMOTE]	0.7824	0.7680	0.8101	0.7882	0.8012
ELM (ReLU) [SMOTE]	0.7778	0.7622	0.8074	0.7840	0.7978
Decision Tree (Gini) [Tomek]	0.8050	0.6840	0.6065	0.6416	0.6198
Weighted Logistic Regression [Tomek]	0.7815	0.5926	0.7752	0.6715	0.7300
Logistic Regression [Tomek]	0.7992	0.6309	0.7327	0.6777	0.7096
Random Forest [Tomek]	0.8141	0.7045	0.6102	0.6536	0.6268
ELM (ReLU) [Tomek]	0.8071	0.6951	0.5902	0.6380	0.6083

Figure 25. 5-fold Cross-validation Scores

9.1.1 Performance Evaluation

From figure 24/25 we can observe models trained on the **SMOTE** resampled dataset perform much stronger in terms of **recall** and **F2 score.** Weighted Logistic Regression with SMOTE has the highest recall of 87% showing strong sensitivity to churners. However, this came at the cost of low precision. I believe the cause was due to the use of class weight when SMOTE already balanced dataset. It caused an extra emphasis on the minority class, thus causing the model to become overly aggressive in predicting churn. It is further clarified by Logistic Regression showing a more balanced trade with its precision and recall scores and maintained a high F2 score. Furthermore, results from the original dataset such as **Random Forest** and **ELM**, achieved higher precision but suffered from lower recall, suggesting they were more conservative in predicting churners. These results highlight how beneficial resampling techniques are in minority detection.

Furthermore, my cross-validation scores suggest strongly, SMOTE resampled dataset generalised well to unseen dataset, and this is clear by similar scores to the test set. On the other hand, Tomek based model, and original methods were not as consistent. One thing to note is that my precision scores where consistently higher during cross validation, this implies a possible sampling skew in the test set or class distribution imbalance that affected precision disproportionately. Therefore, reinforcing the importance of utilising cross-validation to evaluate a model's performance. To conclude **Logistic Regression with SMOTE** would be the most suitable in real world business context. With further possible tuning and validation it would be a suitable candidate to be deployed and used.

Chapter 10: **Software Engineering**

10.1 Methodology

I have utilised an iterative development methodology inspired by the agile approach. This approach means I have broken down the project into smaller and manageable parts, such as experimental data analysis, data preparation, creating models, and model evaluation on my churn dataset. Each phase informed the next, creating a cohesive development pipeline. An example would be the information gained from explorative data analysis of missing data and correlated features, which strongly influenced my decisions in the data pre-processing stage.

Furthermore, I have employed an agile testing strategy by conducting continuous tests. These tests ranged from unit testing for individual classes to testing models on smaller, well, benchmarked datasets.

10.2 Git

Git was essential to the development of my project. It allowed me to manage my code, and since I was utilising RHUL GitLab, it also allowed my supervisor to access my repository for review and feedback easily. I have also followed recommended techniques for version control, such as utilising separate branches in my development, which is evident on my GitLab project page. It ensured encapsulation for developing and testing different models, creating a cleaner workflow. I have also incorporated the use tags and release branches, this marked key milestones in my project from notebook to a full-fledged working Gui.

Furthermore, I followed a conventional commit format of CommitLint[12] for my commits. This made understanding and filtering through git commits much easier and possibly helped if I needed to cherry-pick any commits.

10.3 Test Driven Development

TDD is the methodology I have used to ensure that my models and classes work well. The inbuilt Python unittest module was utilised to ensure that individual components of my project work, specifically my model and evaluation metrics classes. Since I am implementing the machine learning algorithms from scratch, ensuring they are working correctly was key. Therefore, I have compared my scratch model and well-developed SKLEARN [13] existing models on a small dataset. The results from the comparison allowed me to check for any abnormal results and the need for further individual testing of any models.

10.4 Documentation

Documentation is key for many projects as it aids readers in understanding the functionality of code and why certain design decisions were taken. There are many types of documentation, such as comments, UML diagrams to visualise the system's architecture, and external guides or README files. I have included detailed comments throughout most of the codebase, a README and a requirement text file to list relevant dependencies. This ensures clarity and would aid other developers in understanding the functionality of my codebase and aid users trying to run the code.

10.5 Design patterns and GUI

Initially my project was all done Jupyter notebooks and quickly come to realise it's not user friendly as it doesn't separate the code from outputs. For my project to be user friendly I have developed a Gui and utilised 2 main design patterns: MVC, Observer

Model-View-Framework Controller:

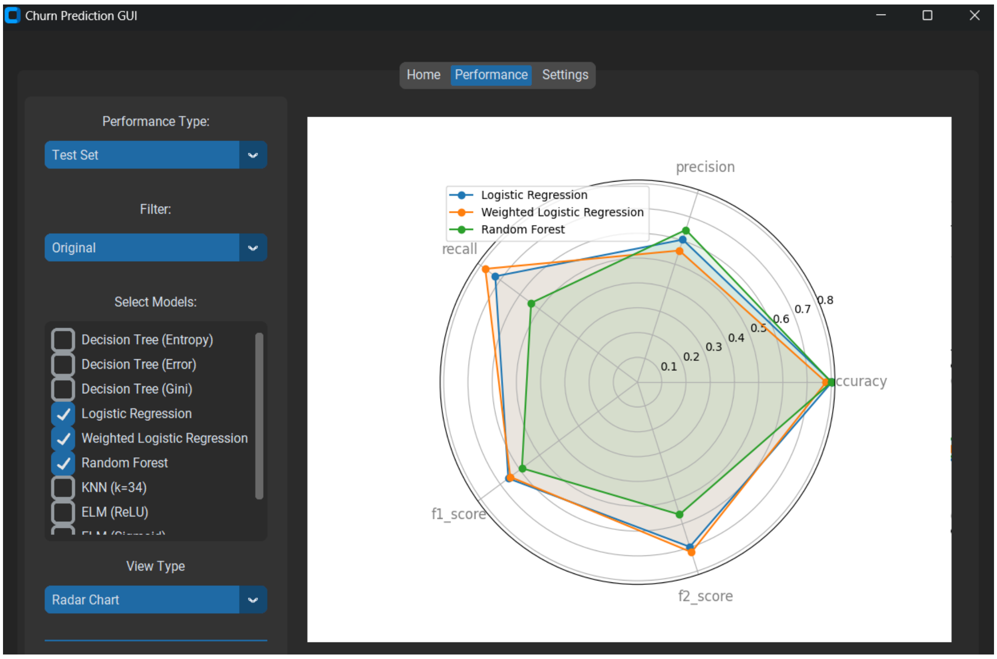
The GUI is built upon the MVC design patterns reducing coupling.

- **Model :** AppModel class handles the loading of models and cross-validation scores from the config files. It then utilises the saved ML models to compute the performance metrics on the test set. Furthermore, the class implements the observer pattern which deals with changes of state in the Gui such as loading of test set performance.

- **View: AppView** encapsulates the front-end aspect of the Gui, this includes the homepage, the performance page and settings page. I have utilised custumtkinter[21] over default Tkinter for a more enhanced and custom design. The view would listen for updates and dynamically update the UI to reflect these changes. An example would be user changing models and selecting a different visual plot.

- **Controller: AppController** is the coordinator of the Gui, handling the intermediary actions between user actions (e.g pressing compare button). The controller is also crucial for the observers as it registers the view (AppView) as an observer of the model during initialisation. This ensures that any changes to the model data automatically triggers an update in the Gui view.

Observer Pattern: The observer pattern was crucial in decoupling the model and the view. Having the models maintain a list of observers allows for quick and dynamic updates whenever there is a change in state of the Gui (e.g a new model added to config). This ensured that there was no tight coupling in the MVC architecture and allowed me to further develop my GUI in later stages with much more ease.



Performance Type: Test Set

Filter: Original

Select Models:

- □ Decision Tree (Entropy)
- □ Decision Tree (Error)
- □ Decision Tree (Gini)
- ☑ Logistic Regression
- ☑ Weighted Logistic Regression
- ☑ Random Forest
- □ KNN (k=34)
- □ ELM (ReLU)

View Type: Radar Chart

Figure 26. Performance tab of the gui

Chapter 11: **Professional Issues**

11.1 Licensing

In a rapidly advancing technological environment, rights of use and property ownership for software and datasets have become critical. Licensing has become the staple of ensuring intellectual property rights and governing and protecting the interests of creators. Many licenses, such as **MIT, Apache 2.0, and GPL**, clarify how software can be used and in what manner. These licensing options aid developers or users of such software in deciding what is a suitable option, whether to make a profit from a closed-source project using such software or collaborative open-source software. Regarding my project, careful dataset selection was challenging due to licensing. I needed to ensure the dataset I selected supported usage for my final project, which is non-commercial and academic-focused. For this reason, I have selected to use a **Kaggle IMB Telecoms** dataset[7]. The Kaggle licensing terms and service allow the usage of publicly shared datasets for academic and research purposes, a perfect fit for my criteria.

11.2 AI-Driven Decision Making

A diverse range of professional issues arise as businesses increasingly rely on artificial intelligence (AI) for strategic and financial decisions. In particular, my project has extensively explored predictive modelling in a churn-based environment, highlighting the significant implications these models hold for retention strategies.AI has become an influential tool within businesses, offering considerable improvements in customer relationship management, decision accuracy, and increased operational efficiency. Nevertheless, the deployment of AI technologies raises critical issues and ethical challenges that must be addressed responsibly and thoughtfully.

In my project, predictive machine learning algorithms were explicitly applied to forecast customer churn. Throughout this process, I assessed multiple algorithms, comparing their effectiveness using performance metrics such as recall, precision, and the F2 score. Although the technical goal was primarily to identify customers likely to churn accurately, the project also brought to light professional issues regarding how these algorithms could realistically influence strategic business decisions.

With businesses becoming ever so reliant on predictive modelling, utilising the right metrics to assess ML models was vital. The usage of just accuracy as the sole evaluation metric was very damaging and misleading. It is solely due to the imbalanced nature of customer data where class distribution is often quite skewed. To combat this issue, extensive metrics such as F1, recall, and precision scores were utilised for a more comprehensive and statistically sound analysis of model results. It allows for a detailed breakdown of the number of false positives, false negatives, and true classifications, offering a more comprehensive understanding of model behaviour beyond just accuracy.

However, these metrics led to another professional Issue. Initially, prioritising recall appeared beneficial, as capturing as many potential churners as possible could reduce revenue loss. However, closer inspection revealed a key drawback: maximising recall without considering precision significantly increased false positives. Practically speaking, this would result in mistakenly categorising loyal customers as likely churn candidates. Such an outcome might lead to unnecessary and potentially counterproductive customer outreach, damaging customer satisfaction and trust.

This reveals an important challenge within AI decision-making. Over-reliance on automated predictions without sufficient human oversight can lead to unintended and problematic outcomes. While AI algorithms undoubtedly enhance efficiency, their recommendations must be assessed with caution to prevent misguided strategic decisions. In my project, parameter tuning was crucial for addressing this balance. I deliberately adjusted model parameters to achieve a practical compromise between recall and precision. This approach ensured that resources would be effectively targeted without negatively affecting customer relations.

Beyond technical issues related to performance metrics, broader professional concerns emerge around organisational reliance on predictive technologies. For instance, excessive dependence on predictive systems may diminish the importance of human judgment and intuition, which remain vital in unpredictable scenarios. Businesses that depend too heavily on machine-generated predictions face difficulties adapting to unexpected changes, as predictive models rely predominantly on historical patterns.

Additionally, ethical considerations surrounding these predictive models must be acknowledged. Machine learning algorithms may unintentionally incorporate biases or replicate inequalities inherent in the data used to train them. Subtle data biases can unintentionally disadvantage specific customer segments even within a neutral-seeming context, such as predicting customer churn. Consequently, aims of transparency, fairness, and explainability were central guiding factors throughout my project. This is evident from my usage of precision – recall graph used to evaluate trade-offs between precision and recall.

Finally, accountability represents another critical professional consideration in AI decision-making. Predictive models often guide essential business actions, including customer retention strategies and targeted marketing initiatives. However, accountability becomes complicated when determining responsibility for erroneous recommendations made by an AI system. Clearly defined accountability mechanisms are essential for maintaining organisational responsibility and trust in automated systems, ensuring businesses remain accountable for AI-influenced decisions. By addressing these issues and adopting appropriate safeguards, I believe organisations can responsibly harness AI's capabilities to support strategic decisions.

Chapter 12: **Self-Reflection**

12.1.1 Self-Reflection

Taking a moment to self-reflect, I am proud of what I have achieved and the hurdles I have overcome. While my ML project was not as flashy as Neural Nets and Deep learning, which currently dominate the field, this benefited my growth and understanding. Building ML models from scratch ensured my understanding and allowed me to engage deeply with the theory behind the algorithm.

Furthermore, choosing the domain of customer churn mimics the simulation of real-world jobs. It has taught me a lot, from conducting explorative data analysis thus informing my data preprocessing to building an ML pipeline for the GUI. It taught me that machine learning is more than selecting the most powerful model. It is about understanding your data and carrying out investigations to make well-informed decisions to benefit your end goal. These lessons will stick with me and aid me in my professional career. I would also like to thank my supervisor, **Dr Anand Subramoney,** for his help and guidance.

12.1.2 Possible Improvements

Given that I achieved most of what I planned out except for the minor tweaks in my plans, there are still possible improvements that I could add to my project.

- Extend the GUI for user to input their own test data and get real time predictions
- Investigate feature engineering and its impact on performance
- Integrate more ensemble models such as AdaBoost
- Add confidence measures to prediction outputs
- Investigate the effects of combining SMOTE and TOMEK-Links to churn dataset
- Allow for performance results in the GUI to be downloadable

Appendix: Diary

Project Diary

Date: 1st October 2024

Summary

• Today, I concentrated on determining the specific sub-problem I will focus on for comparing ML algorithms. After some consideration, I decided to pursue churn prediction.

• I also began evaluating which datasets i would be using in this project. In contention between Telecom’s data set and Bank dataset.

Next Steps

• Continue researching and start writing up a project plan

• Research some basic machine learning models, such as nearest neighbours, to establish a foundational understanding.

• Continue exploring potential datasets for benchmarking and implementation in the churn prediction model.

Date: 4th October 2024

Summary

• I looked into research papers on machine learning algorithms and churn prediction to gain a wider understanding and understand on how to approach my fyp.

• Initial idea i have is to have multiple different ml models to be used to predict churn such as decision tree, Logistic regression , support vector machines. Then to evaluate them empirically and give some findings on each model.

Next Steps

• Continue researching and writing up the project plan

• Discussed project plan and my churning sup problem with the supervisor before the project plan deadline.

Date: 8/10/24

Summary

•Completed the first draft of the project plan. I also created a set of question to be asked for supervisors meeting

Date: 10/10/24

Summary

•I met with my supervisor, discussed the project plan, and got feedback on my questions. I then went on to finalise the project plan and submit it.

Date: 14/10/24

Summary

- I have selected the telecom dataset for my project and have been researching exploratory data analysis (EDA) to understand its importance..

- I plan to carry out EDA on the dataset, and in the meantime, I have been learning about the Seaborn library and how it can be utilised effectively for visualising data during EDA.

Date: 19/10/24

Summary

- I have begun conducting exploratory data analysis (EDA) on the telecom dataset.

- I carried out observations on the distribution of missing data using a heatmap.

- Might run into issues later, since there are a number of rows with missing data.e.g internet usage and device protection. This would mean it would be harder to establish credible relationships between features hindering churn prediction performance potentially

Date: 21/10/24

Summary

Today, I continued Exploration Data analysis on the Telecom dataset. I utilised the Seaborn library to observe the distribution of churn categories visually and made my observations. Similarly, I made similar observations for other continuous variables, plotting histograms to understand the data better.

However, given that the dataset contains quite a lot of missing data values, I'm planning on changing datasets and researching alternative datasets that might offer more complete information. It would mean I would have to redo EDA, but it should be easier now.

Date: 22/10/24

Summary

I have selected a new dataset by the company IBM that can be downloaded from Kaggle. The dataset is a telecom dataset, and I have started EDA again. It was much easier this time as I had good practice with the previous dataset. The dataset did not have much missing data, and we dealt with it by deleting those rows. I had the choice of using the median values of the row to replace the missing data but opted to delete those rows as it was a deficient number(11 rows). I ran into some issues with the types of some of the dataset as some where object when it should have been float. Made me realise how important EDA is as it prevents errors later on when building Ml models.

Date: 26/10/24 - 30/10/24

Summary

Over these last few days, I decided to get my EDA completed, since the goal is to move on to actually implement ML algorithms that predict churn. First, I started investigating which customer attributes might have decent or strong correlation to churn. For each attribute, visual diagrams were created, such as boxplots and histograms, in order to discover some underlying relations between attributes.

I also did deep analysis with Demographics and Payment Types. From the customers, whether SeniorCitizen and Partner, based on these two statuses, I found some trends in the Churn behavior. Then, I moved my attention to features regarding service. I analyzed features like Online Security, Tech Support, Phone Service, and Multiple Lines to determine whether the kind of services provided had some impact on the churning rate. This helped me highlight customers who could be in danger of churning because of a lack of additional services. I also explored Internet Service types and Payment Methods to see if there are any service types or payment preferences that could be related to churning. An example of observation i made was that the customers using an electronic check as a payment method showed higher churn. Based on that it appears there is some relationship to the billing types and customer satisfaction.

Having completed the EDA, it gave insight into the structure of the data and relationships that existed. That allowed me to get an idea of what features may play an important role in predicting churn. That would help me with the next part of my development, building ML models and feature engineering.

Date: 10/11/24

Summary

I have been preoccupied the last few days with assignments and didn’t get to do as much work as I liked on my final year project, and now I need to catch up on my timeline. Today, I began my data preprocessing stage. I realised a mistake in my Explorative Data Analysis, where I deleted 11 rows of data because the total charge was missing. While this is partially correct, each data object has valuable information that can aid churn prediction modelling. To combat this, I utilised how long each customer stayed with the company and their monthly charge to manually compute the total charge.

After this, I decided to investigate how to convert the data for ML model as some of the data was non-numerical, such as payment methods. This is where I then learnt about encoding data and the various types of encodings. Given my dataset, I utilised binary encoding for binary variables (e.g., Yes/No or Male/Female categories) and one-hot encoding for categorical variables with multiple unique values (such as PaymentMethod or InternetService). Overall, today I have set a solid foundation for model training and will be moving on to normalisation/scaling next which is key for my first ML algo KNN.

Date: 11/11/24

Summary

Today, I completed my data preprocessing process. I did this by normalising the data and creating 2 different normalised versions of the dataset. One is the min_max normalised date, while the other is the z_score dataset. I did this to test which normalised dataset performs better with my KNN model. After this, I began developing the KNN model from scratch, utilising no libraries and using Euclidean distance as the distance metric. My initial implementation was working very slowly. This was mainly due to how I was sorting computing distances utilising insertion sort, which has a time complexity of O(n^2), which could be better given that my dataset was 7000+.

After some research, I learnt about heapq, which has a time complexity of O(logn). This reduced my computation time from 20 minutes to about 1-2 minutes, which is a huge accomplishment. I also learnt how important optimisation is with ML models.

Date: 12/11/24 - 15/11/24

Summary

I began testing out the KNN models I created on my dataset. Initially, I ran my KNN with k set to 1 neighbour and tested it on both min_max and z_score normalised churn datasets. The results were relatively okay, with an error rate of about 0.23, which is understandable since KNN is a lazy algorithm. I then ran my algorithms on larger numbers of K to find the optimal k and plotted a bar chart of error results.

here the optimal k value is 34 due to having lowest error rate.

However, I had some slight issues of rerunning models each time I open up my notebook which would be time consuming. I began investigating ways to save models such that i would not require retraining each time. This is where I found pickle, a Python library that allows saving and reloading objects such as models or evaluation metrics. This allowed me to focus on interpreting the results and visualising the error rates without redundant computations especially when testing different configurations or normalisation methods. Overall, my workflow has become more optimised compared to before.

Date: 18/11/24

Summary

After this, I, then decided to implement a known conformal predictor. This allowed me to quantify uncertainty and assign confidence levels to predictions. However my implementation could be more efficient as it takes a long time to run on a large dataset so I have decided to test on the subset of the dataset(3500). This required me to go through Vladimir Vovk's research work on conformal predictors, which was interesting.

My implementation was a bit crude and could be improved with some optimisation as the run time was very long, and this could be due to repetitive copying of conformity score. However, I did improve its optimality by only recomputing conformity score when necessary, thus reducing run time greatly from previous runs.

Date: 20/11/24

After some research on evaluation metrics for classification, I have realised that the current metrics I'm using, accuracy and error rate, are very weak. So, I have decided to use the metrics F1-score, accuracy, recall, precision, and F2-score. I created these from scratch, utilising the knowledge gained. This would help a lot since my dataset is imbalanced and give further insights, such as the number of false negatives and false positives. Utilising my new metrics class, I then updated my main Jupyter notebook to have much more in-depth results and added graphs. Overall, I am making some progress, but I should investigate/research more before coding or developing so that I wouldn't have to refactor too much.

Date 22/11/24

I have researched more about software engineer methodologies in ML and based on that I have decided to implement unit test cases for classes which is what I have done. These test classes were added for conformal and KNN classes and are fully working.

My knn is model is fully complete now and have merged into main branch with no issues.

Date 26/11/24

I have been watching videos and looking into papers about decision tree models in the recent days. I specifically focused on different uniformity measures, utilising this knowledge I then implemented a decision tree class, haven't fully tested it on my data but will do that soon. I initially started with a basic Decision tree class but realised to accommodate for different uniformity measures I would have to split into 2 classes and adjust the parameters. Looking at my original plan I have overestimated the amount of work I could do, this mainly due to workload of other modules. I am a bit behind but not so much

I have also booked a meeting with my supervisor again tomorrow and this is mainly to establish if I'm on track and to answer any questions I have.

Date 27/11/24

Today I attended my meeting with my supervisor, and I have completed a good amount of work for interim submission. I also managed to discuss what should be in the report and how I could go about presenting my FYP later on.

Furthermore, I have carried out initial testing for my Decision tree class and is preforming relatively well, some higher scores compared to knn.

Date 05/12/24 - 09/2/24

I have been really sick these past couple of days. It is a shame, considering I wanted to improve the validity of my models by adding cross-validation and perhaps adding decision boundary results for my different models. This has set me back as now I need to complete my report writing; this also meant I don't think I would be able to do my SVN model or basic GUI. However, on the bright side, I have identified and fixed a key issue; this was the fact that I was normalising the whole dataset and then splitting the churn dataset for testing and training. This had led to data leakage, which affected the validity of my results. I fixed this issue by splitting and then normalising my dataset. This has led to slightly lower results in my decision tree but hasn't affected my KNN model results.

Furthermore, in this time period I have finished my testing of decision tree model and am currently writing up my report/presentation.

Date 27/01/25

I have scheduled a meeting with my supervisor to discuss feedback on my interim submission. Overall, it was positive feedback on my interim submission, and it identified key issues with my report that I will fix. We also discussed moving away from working on Jupyter notebooks and into a working GUI with the models. We also discussed some possible extensions to the project, such as ensemble methods and investigating imbalance methods.

Date 04/02/25

I have spent time refactoring my project directory so that it meets Pep8 compliance, as my supervisor suggested. I also spent some time deciding on what front end I would use for the GUI, in between Flask and Tkinter. I also fixed issues with the report, namely the bibliography.

Date 10/02/25 - 13/02/25

I am working on moving all the code from the notebooks into standalone packages and folders. This meant redoing preprocessing by creating a separate script. All the separate pieces are slowly coming together, which would be vital for the GUI

Date 19/02/25 - 24/02/25

I am a bit behind schedule due to refactoring old code for future GUI development. Also, after refactoring, I needed to implement cross-validation on my existing models. This required some time as my custum models didnt work with sklearn methods. Needed to understand the coding logic required to implement my own version. Ran into quite a few issues when moving code away from notebook namely folders and models not recognising imports which was tedious to fix. My existing models incorporate CV, but I need to utilise it more effectively.

Date 26/02/25

I have finsihed setting up for my gui development and now started working on a gui. My mine idea is a dashboard where users can select different models and different visualisation options. Also, I have been researching Software engineering and design patterns for gui development. I have selcted tkinter as my gui design langauage as its much easier to integrate with my exsitng models and dataset. Learnt about observer and MVC design patters which makes development and structure much cleaner and reduces compelixty/coupling alot.

Date 27/02/25 - 03/03/25

So I have finally implemented an early working version of my gui. Its a very early version, so far it allows user to select 2 different models KNN and DT models. Once user selcted models they are able to visually see test results i.e by selecting bar chart. Had some difficulties managing to form the pipeline of models to training and then loading for gui. Managed to implement MVC pattern and observer pattern into the gui. I need to fix the hardcoded filepaths and also improve gui layout and design

Date 04/03/25 - 05/06/25

I managed to improve the pipeline for the GUI integration, doing this through dedicated scripts for training and loading. Now, the backend side of the GUI is fully working as it should. I also managed to remove hardcoded file paths and replace them with config files. I began working on my random forest implementation and built upon it using my DT class.

Date 06/03/25 - 10/06/25

I implemented my random forest model and verified its effectiveness by comparing it with Sklearns on a small dataset. I need to make some minor tweaks for it to be integrated correctly with my GUI.

Date 11/03/25 - 14/06/25

I have been researching and understanding how Logistic regression works. I focused on the key functionalities needed to implement it. The core functionalities are working, and my results are not too bad. I added L2 reg to further improve my results, but it didn't make too much of a difference.

Date 16/03/25 - 18/03/25

I had a meeting with my supervisor last week; my progress is not bad, gave me some good advice on how I can structure my report and what further models I should investigate. So far, I have been working on a weighted version of logistic regression and Extreme learning machines suggested by my supervisor.

Date 18/03/25 - 20/03/25

I have been re-looking at research papers on resampling for imbalanced datasets that i have saved. The two main ones I focused on are SMOTE and TOMEK links. I have implemented SMOTE and aim to implement Tomek links as well. So far, I have seen some promising results with SMOTE; noticeably, my recall scores have increased, which is essential given the context of my problem.

Date 20/03/25 - 02/04/25

I have managed to complete my imbalance resampling scripts and methods. These were Tomek links and SMOTE; they were fully tested and, thinking about the GUI developments, have created relevant scripts to separate original and resampled datasets. I have begun planning out my final report, and the next stage of development is ELM and further gui work.

Date 03/04/25 - 04/04/25

I have made a lot of progress and am nearing the end goal for the coding side of my project. I have created candidate release branches. Taking the advice from my supervisor, I have implemented the Extreme Learning machine model and integrated it into the GUI. I have been working on the GUI a lot and have carried out some hyperparameter tuning.

Date 06/04/25

My coding side is nearly complete, the GUI is fully working, and I have added a new release branch. I have had quite a bad headache but have been managing to power through. I have also been working on the report simultaneously and am a bit behind, but it should be manageable.

Date 07/04/25

I managed to get an extension/late waiver and have informed my supervisor; this gives me some leeway to recover from my illness and ensure I achieve my goals for my final-year project. My code base is in good condition; I just need to finish up my report writing.

Bibliography

[1] Oliver Wyman. Customer churn, 2004.

[2] Outsource Accelerator. Crucial customer retention statistics to know in 2024, 2024.

[3] Sabrina Tessitore. What’s the average churn rate by industry? CustomerGauge, 2022.

[4] F.F. Reichheld and P. Schefter. E-loyalty: Your secret weapon on the web. Harvard Business Review, 2000.

[5] D. Gefen. Customer loyalty in e-commerce. Journal of the Association for Information Systems, 2002.

[6] J. Morrison and A. Vasilakos. An empirical comparison of techniques for the class imbalance problem in churn prediction, 2018.

[7] IBM / Kaggle (Blastchar). Telco Customer Churn Dataset, 2017.

[8] I.H. Witten, E. Frank, M.A. Hall, and C.J. Pal. Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann, 2016.

[9] M. Kuhn and K. Johnson. Applied Predictive Modelling, 2013.

[10] P. Cunningham and S.J. Delany. k-Nearest neighbour classifiers - a tutorial. ACM Computing Surveys, 54(6):1–25, 2021.

[11] J. Han, M. Kamber, and J. Pei. Data Mining: Concepts and Techniques. Morgan Kaufmann, 2011.

[12] CommitLint. CommitLint documentation.

[13] Scikit-learn. Supervised learning.

[14] D. Jurafsky and J.H. Martin. Speech and Language Processing (3rd ed. draft), Chapter 5: Logistic Regression, Stanford University, 2023.

[15] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew. Extreme Learning Machine: Theory and Applications. *Neurocomputing*, 2006.

[16] Ben-Israel, A., and Greville, T.N.E. Generalized Inverses: Theory and Applications. 2nd ed., Springer, 2003.

[17] H. Chen, A. G. Cohn, D. Zhang, and M. Zhou, “Machine learning-based classification of rock discontinuity trace: SMOTE oversampling integrated with GBT ensemble learning,” *International Journal of Mining Science and Technology*, vol. 32, 2021.

[18] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “SMOTE: Synthetic Minority Over-sampling Technique,” *Journal of Artificial Intelligence Research*, vol. 16, 2002.

[19] G. Batista, R. C. Prati, and M. C. Monard. A study of the behaviour of several methods for balancing machine learning training data. *SIGKDD Explorations*, 2004.

[20] Towards Data Science. *How Tomek Links Work*, 2020.

[21] Tom Schimansky. *CustomTkinter Documentation*. CustomTkinter Project, 2025.