CS 178: Final Project

FRONT COVER

CS 178 Final Project

Names	**Mohammadarshya Salahibakhsh, Vishok Lakshmankumar, Eric Tao**
IDs	**71713160, 88285812, 75725545**
Dataset	**Diabetes 130-US Hospitals**
Classification Methods	**KNN, Logistic Classifier, XGBoost, MLP**

Dataset: Diabetes 130-US Hospitals
Classification Methods: KNN , Logistic classifier, XGBoost, MLP

Authors: Vishok Lakshmankumar, Mohammadarshya Salahibakhsh, Eric Tao
Team Name: EVA(Diabetes)

Date: 06/07/2025
Collaborators: (List staff members/externals who have helped)

Summary

In this project, we investigate machine learning classification methods to predict hospital readmission within 30 days using the Diabetes 130-US Hospital dataset. We preprocess the dataset, handle missing values, and apply several classification models including k-Nearest Neighbors (kNN), logistic regression, multilayer perceptron (MLP), and XGBoost. Our results show that tree-based and neural classifiers outperform simpler methods, with MLP and XGBoost achieving the highest accuracy and robustness across training sizes.

Data Description

We use the **Diabetes 130-US Hospital dataset**, a clinical dataset containing over 100,000 encounters across 49 features including patient demographics, diagnoses, and medications. Our target variable is hospital readmission within 30 days (<30), operationalized as a binary label (1 if readmitted within 30 days, 0 otherwise).

Exploratory Steps:

1. Dropped ID-like columns (encounter_id, patient_nbr), irrelevant or high-missing-value columns (weight, payer_code, medical_specialty).
2. Replaced "?" with NaN.
3. Created binary target readmitted_binary.

Summary: (+) -> positive, (-) -> negative

Total Sample	(+) Class count <30	(-) class count	Number of Features	Numeric Features	Categorical features	Percent missing values
101766	11357	90409	44	11	33	3.96

Class Distribution: Readmitted <30 days: ~11% and Not readmitted: ~89%

See **Figure 1** for the distribution across classes and **Figure 2** for a heatmap of the features' missing values.

Related Work:

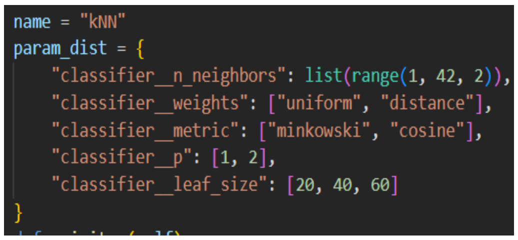
1. *"Predicting 30-Day Hospital Readmission in Patients With Diabetes"* (2024)

- **Overview**: A retrospective study comparing logistic regression, random forest, XGBoost, and deep neural networks to predict 30-day readmissions.
- **Usage of Dataset**: Explicitly uses the UCI Diabetes 130-US Hospitals dataset
- **URL**: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC12085305/ pmc.ncbi.nlm.nih.gov

Classifiers

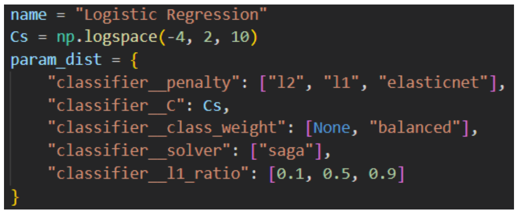
We implemented and compared four classification models: **k-Nearest Neighbors (kNN)**, **Logistic Regression**, **Multilayer Perceptron (MLP)**, and **XGBoost**.

a. k-Nearest Neighbors (kNN) Description and Hyperparameters investigated:
kNN is a simple yet powerful classifier that assigns a label to a sample based on the labels of its k nearest neighbors. It is memory-based and does not build an explicit internal model. **Software**: Scikit-learn (KNeighborsClassifier)



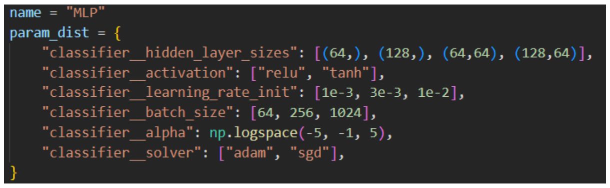
```
name = "kNN"
param_dist = {
    "classifier__n_neighbors": list(range(1, 42, 2)),
    "classifier__weights": ["uniform", "distance"],
    "classifier__metric": ["minkowski", "cosine"],
    "classifier__p": [1, 2],
    "classifier__leaf_size": [20, 40, 60]
}
```

b. Logistic Regression Description Hyperparameters Investigated
Logistic regression models the probability of the default class using a sigmoid function. It works best on linearly separable data. In this project, we applied different regularization schemes to handle overfitting. **Software**: Scikit-learn (LogisticRegression with max_iter=1000 and multi_class='ovr')



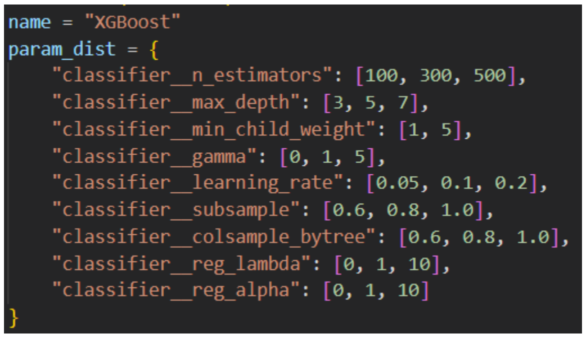
```
name = "Logistic Regression"
Cs = np.logspace(-4, 2, 10)
param_dist = {
    "classifier__penalty": ["l2", "l1", "elasticnet"],
    "classifier__C": Cs,
    "classifier__class_weight": [None, "balanced"],
    "classifier__solver": ["saga"],
    "classifier__l1_ratio": [0.1, 0.5, 0.9]
}
```

c. Multilayer Perceptron (MLP) Description and **Hyperparameters Investigated**
An MLP is a class of feedforward artificial neural networks that consists of an input layer, one or more hidden layers, and an output layer. It is capable of learning non-linear decision boundaries by applying multiple layers of transformations using activation functions. **Software**: Scikit-learn (MLPClassifier with early_stopping=True and max_iter=300)



```
name = "MLP"
param_dist = {
    "classifier__hidden_layer_sizes": [(64,), (128,), (64,64), (128,64)],
    "classifier__activation": ["relu", "tanh"],
    "classifier__learning_rate_init": [1e-3, 3e-3, 1e-2],
    "classifier__batch_size": [64, 256, 1024],
    "classifier__alpha": np.logspace(-5, -1, 5),
    "classifier__solver": ["adam", "sgd"],
}
```

d. XGBoost Description and **Hyperparameters Investigated**
XGBoost (Extreme Gradient Boosting) is an optimized gradient-boosting algorithm that builds a group of shallow decision trees to sequentially reduce prediction error. It is known for high performance, fast training speed, and handling sparse/missing data effectively. **Software**: XGBoost Python library (XGBClassifier with tree_method='hist', eval_metric='logloss', and use_label_encoder=False)



```
name = "XGBoost"
param_dist = {
    "classifier__n_estimators": [100, 300, 500],
    "classifier__max_depth": [3, 5, 7],
    "classifier__min_child_weight": [1, 5],
    "classifier__gamma": [0, 1, 5],
    "classifier__learning_rate": [0.05, 0.1, 0.2],
    "classifier__subsample": [0.6, 0.8, 1.0],
    "classifier__colsample_bytree": [0.6, 0.8, 1.0],
    "classifier__reg_lambda": [0, 1, 10],
    "classifier__reg_alpha": [0, 1, 10]
}
```

Experimental Setup

To evaluate our classifiers in a reproducible and unbiased manner, we followed a structured experimental protocol involving data partitioning, pipeline preprocessing, hyperparameter tuning, and evaluation using multiple metrics.

Data Partitioning

We divided the dataset into three disjoint sets:

- **Training set (60%)** - used to fit models.
- **Validation set (20%)** - used during hyperparameter tuning and model selection.
- **Test set (20%)** - used **only once** at the end of the project to evaluate final performance.

This split was performed using train_test_split() from Scikit-learn with stratify=y to preserve class proportions due to the imbalanced nature of the dataset (only ~11% of cases are positive, so we needed to make sure each subset had both positive and negative cases).

Preprocessing Pipelines

We preprocessed our data using ColumnTransformer:

- **Numerical features**: missing values replaced by mean + standard scaling
- **Categorical features**: missing values replaced by mode + one-hot encoding

This preprocessing was included in each classifier's Pipeline by adding it as an attribute of our custom classifier classes, ensuring consistent data handling during both training and evaluation.

Hyperparameter Tuning

We used RandomizedSearchCV with 3-fold cross-validation to efficiently search through each model's hyperparameter space so that we did not have to train on every combination of hyperparameters to reduce training time. Each classifier's class has a param_dist attribute that contains all hyperparameter values to be trained on. We sampled 5 random combinations of hyperparameters for each classifier (n_iter=5). By using random_state=178, this random sampling is reproducible across multiple runs. The hyperparameters with the best accuracy on the validation set were retained and used to train a final model evaluated on the test set.

The results of this first step of tuning and evaluation can be seen in **Figures 3 and 4.**

Evaluation Metrics

Once we found the best parameters for each classifier, we mainly used these three following metrics to assess which classifier would be best for our dataset:

- **Accuracy**: Overall correct predictions on the test set.
- **Learning Curves**: For each classifier, we plotted training/validation accuracy as a function of training set size using Scikit-learn's learning_curve() function.
- **Confusion Matrix**: Used for qualitative error analysis and to derive precision/recall.

We also used **ROC-AUC** and **Brier Score** to verify our findings about dataset imbalance from the precision and recall in the confusion matrix:

- **ROC-AUC**: Quantifies the tradeoff between true positive and false positive rates; higher values indicate better discrimination. This is especially useful to see if there are any issues caused by our data imbalance.
- **Brier Score**: Evaluates the calibration of probabilistic outputs; lower values reflect better confidence accuracy.

This experimental design evaluates rigorously while also properly following machine learning principles to allow for reproducibility. Hyperparameters were optimized on the validation set, and the final reported numbers are from the test set which was never used during tuning.

Experimental Results

We evaluated the performance of the four classifiers using multiple metrics on a held-out test set. Each model was trained with hyperparameters selected through cross-validation, and final performance was assessed on data never seen during training or tuning.

Summary of Test Set Performance

Classifier	Accuracy	ROC-AUC	Brier score
KNN	~86%	~0.60	~0.11
Logistic	~87%	~0.63	~0.10
MLP	~89%	~0.68	~0.09
XGBoost	~90%	~0.69	~0.085

See **Figure 5** for a graphical representation of the models' performances. When just looking at the accuracies, it seems like our models are performing very well. However, once you remember that our data set had an 89% to 11% imbalance between classes and look at the ROC-AUC and Brier Scores as well as the confusion matrices in the next section, it is clear that our classifiers have a major flaw.

Learning Curves

We generated learning curves for each classifier by training them on progressively larger subsets of the data (5%, 10%, 20%, 40%, 80%) and measuring both training and validation accuracy. See **Figure 6** for the graphs of the learning curves. **kNN** had very high accuracy at the start, and did not increase in accuracy as batch size grew, suggesting overfitting. **Logistic Regression** had stable learning, showing steady improvements with data size. **MLP** and **XGBoost** benefited most from larger datasets, with validation accuracy increasing significantly with more data. **MLP** showed some variance early on but generalized well after sufficient data exposure and even exceeded the training accuracy.
These curves illustrate that more complex models like MLP and XGBoost better use the increase in data compared to simpler ones like kNN.

Confusion Matrix and Jaccard Similarity Analysis

For each classifier, we plotted confusion matrices to visualize their breakdown of true/false positives and negatives, which can be seen in **Figure 7**. As seen in the confusion matrices, the models rarely ever output true. All 4 classifiers learned to almost always output false because it is correct 90% of the time because the dataset only has 10% true cases. This leads to a lot of false negatives. This is especially important in healthcare settings where missing a potential readmission case can lead to worse patient outcomes. **Figure 8** shows a Jaccard similarity to confirm this; all 4 classifiers are making the exact same errors. To fix this issue, we retrained the classifiers while optimizing for a recall of at least 20% this time by passing in scoring="recall" to RandomizedSearchCV. Once the recall of 20% was reached, we aimed to maximize precision and then accuracy. This led to slightly lower accuracies than before, but far less false negatives as seen in **Figure 9**, which is more important in clinical practices.

Insights

Based on the results in **Figure 9**, XGBoost outperforms all others in terms of both predictive performance and calibration to allow better recall and precision, making it well-suited for deployment. Logistic Regression is a close second in terms of having good recall and precision, but its accuracy is slightly lower. MLP also offers competitive recall, precision, and accuracy, but it is still slightly worse than XGBoost. KNN, on the other hand, is far too simple and less scalable and lags behind the other models after being optimized for recall. This taught us that model complexity matters in choosing the right classifier for certain datasets. To add on to this, we also learned that data imbalance matters. Since only about 11% of cases had true labels of “true,” the models all learned to just say “false” every time and get a good accuracy. This shows that picking the right dataset is very important when training classifiers. Also, even if you do not have the right datasets, you have to look at advanced evaluation metrics like precision and recall rather than just relying on maximizing accuracy. ROC-AUC and Brier Score were other metrics that helped with this as well. Another thing that was insightful was the learning curve plots; we assumed that adding more data would always lead to more accurate models. However, kNN proved this wrong, as adding more data did nothing for the CV accuracy. This shows that depending on the accuracy of your model, you may get diminishing returns from significant increases in your batch size.

By comparing misclassified examples across classifiers, we noticed several patterns. Cases misclassified by all models tended to involve borderline or conflicting feature values. For example, some patients had short hospital stays but high lab test counts or unusual medication patterns. These difficult cases may even confuse human doctors, especially if the underlying medical records lack context (such as socioeconomic factors or home care availability). Also, we noticed that the classifiers were struggling with cases where there was missing or incomplete data. This does mimic real-world data, as not all patients will have data for every feature. So, it may be safe to assume that our models are too simple to use safely in the real world.

Final Reflection

Overall, the project illustrated the trade-offs between simplicity, performance, interpretability, and robustness in clinical machine learning applications. The most valuable outcome wasn’t just building accurate models–it was learning how to validate and modify those models in a way that would matter in the real world. We also came away with an appreciation for how critical proper preprocessing, evaluation metrics, and experimental rigor are when working with imbalanced and high-stakes datasets like those in healthcare.

Contributions

Mohammadarshya focused on generating visualizations and figures used throughout the report, including class distribution, learning curves, confusion matrices, calibration plots, and error overlap analysis. They also took the lead in writing and organizing the final project report, ensuring that each section was clearly structured and effectively communicated the results and insights.

Vishok focused on the initial model training and creating metrics for evaluating which classifiers would be best. He also created visualizations for the confusion matrices, learning curves, and Jaccard similarity and analyzed them. He also helped determine the best strategies for improving the model.

Eric mainly focused on improving the initial models after we realized that they had very low precision and recall. He researched ways to train the model based on these advanced metrics rather than just simply maximizing the accuracy of the models.

Appendix

Figure 1 - Class Distribution

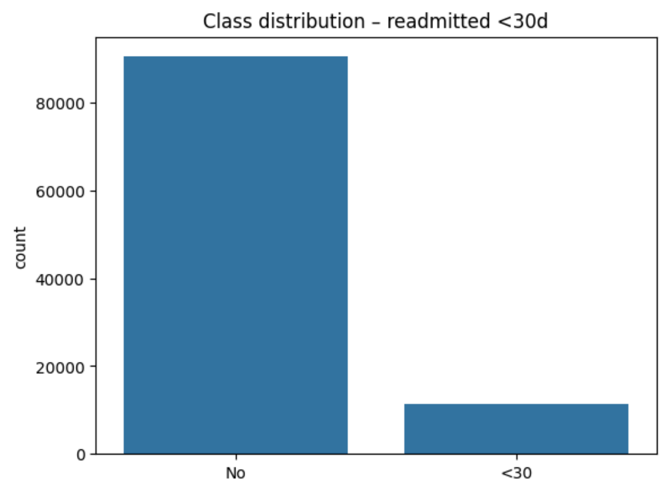


Figure 2 - Missing Values HeatMap (Categorical Features)

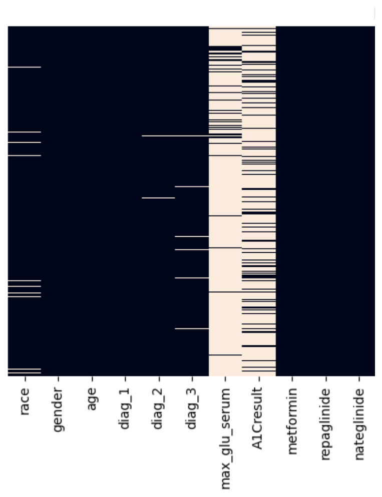
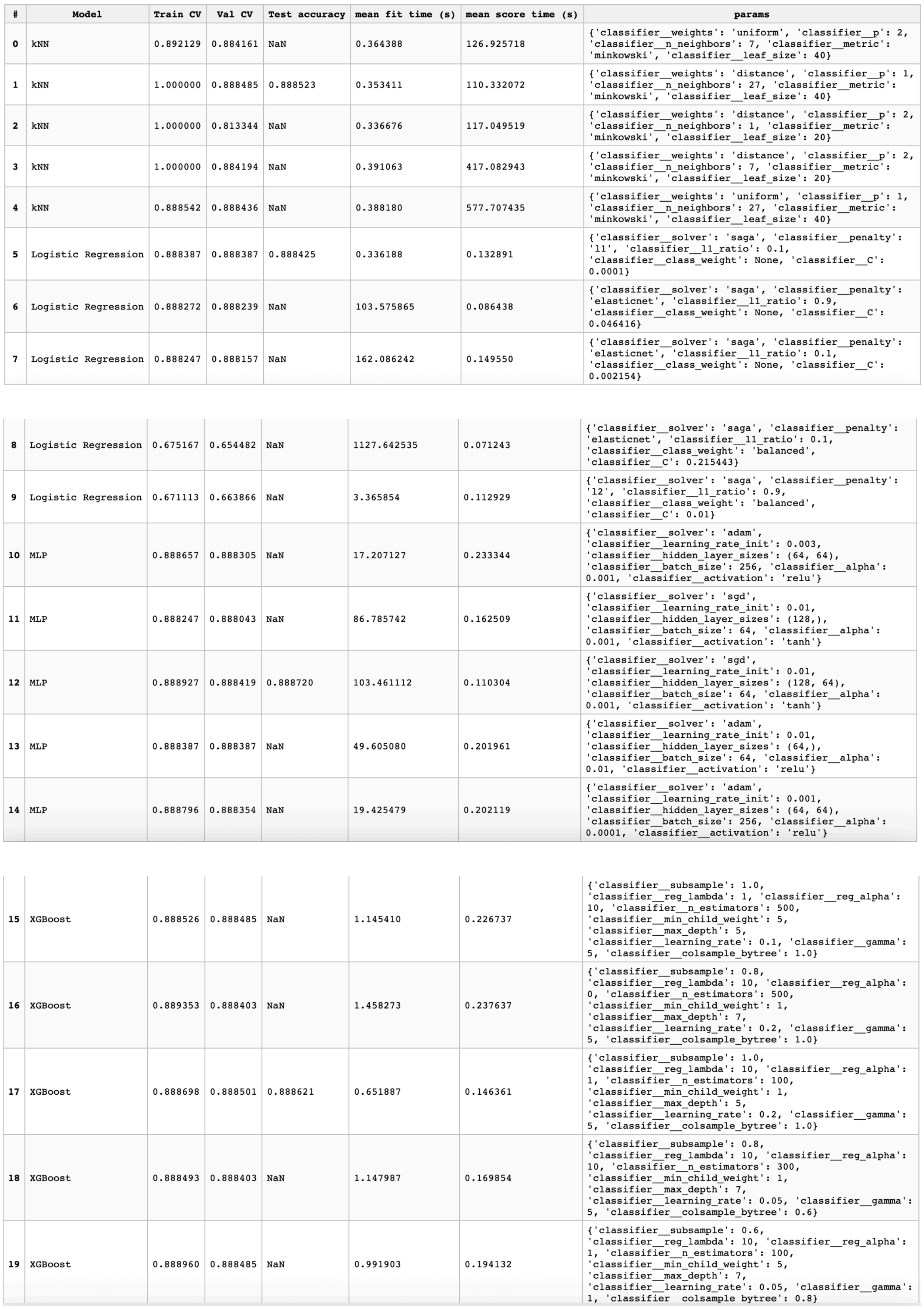
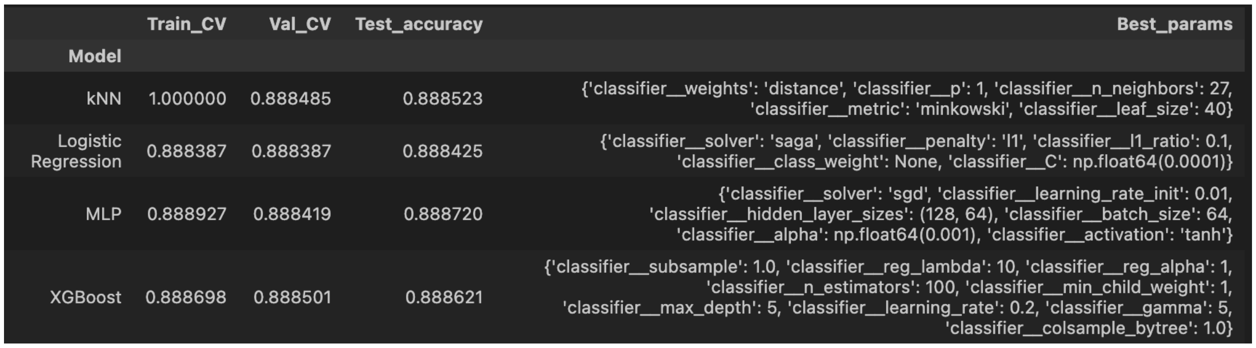


Figure 3 - All Hyper-Parameter Trials



#	Model	Train CV	Val CV	Test accuracy	mean fit time (s)	mean score time (s)	params
0	kNN	0.892129	0.884161	NaN	0.364388	126.925718	{'classifier__weights': 'uniform', 'classifier__p': 2, 'classifier__n_neighbors': 7, 'classifier__metric': 'minkowski', 'classifier__leaf_size': 40}
1	kNN	1.000000	0.888485	0.888523	0.353411	110.332072	{'classifier__weights': 'distance', 'classifier__p': 1, 'classifier__n_neighbors': 27, 'classifier__metric': 'minkowski', 'classifier__leaf_size': 40}
2	kNN	1.000000	0.813344	NaN	0.336676	117.049519	{'classifier__weights': 'distance', 'classifier__p': 2, 'classifier__n_neighbors': 1, 'classifier__metric': 'minkowski', 'classifier__leaf_size': 20}
3	kNN	1.000000	0.884194	NaN	0.391063	417.082943	{'classifier__weights': 'distance', 'classifier__p': 2, 'classifier__n_neighbors': 7, 'classifier__metric': 'minkowski', 'classifier__leaf_size': 20}
4	kNN	0.888542	0.888436	NaN	0.388180	577.707435	{'classifier__weights': 'uniform', 'classifier__p': 1, 'classifier__n_neighbors': 27, 'classifier__metric': 'minkowski', 'classifier__leaf_size': 40}
5	Logistic Regression	0.888387	0.888387	0.888425	0.336188	0.132891	{'classifier__solver': 'saga', 'classifier__penalty': 'l1', 'classifier__l1_ratio': 0.1, 'classifier__class_weight': None, 'classifier__C': 0.0001}
6	Logistic Regression	0.888272	0.888239	NaN	103.575865	0.086438	{'classifier__solver': 'saga', 'classifier__penalty': 'elasticnet', 'classifier__l1_ratio': 0.9, 'classifier__class_weight': None, 'classifier__C': 0.046416}
7	Logistic Regression	0.888247	0.888157	NaN	162.086242	0.149550	{'classifier__solver': 'saga', 'classifier__penalty': 'elasticnet', 'classifier__l1_ratio': 0.1, 'classifier__class_weight': None, 'classifier__C': 0.002154}
8	Logistic Regression	0.675167	0.654482	NaN	1127.642535	0.071243	{'classifier__solver': 'saga', 'classifier__penalty': 'elasticnet', 'classifier__l1_ratio': 0.1, 'classifier__class_weight': 'balanced', 'classifier__C': 0.215443}
9	Logistic Regression	0.671113	0.663866	NaN	3.365854	0.112929	{'classifier__solver': 'saga', 'classifier__penalty': 'l2', 'classifier__l1_ratio': 0.9, 'classifier__class_weight': 'balanced', 'classifier__C': 0.01}
10	MLP	0.888657	0.888305	NaN	17.207127	0.233344	{'classifier__solver': 'adam', 'classifier__learning_rate_init': 0.003, 'classifier__hidden_layer_sizes': (64, 64), 'classifier__batch_size': 256, 'classifier__alpha': 0.001, 'classifier__activation': 'relu'}
11	MLP	0.888247	0.888043	NaN	86.785742	0.162509	{'classifier__solver': 'sgd', 'classifier__learning_rate_init': 0.01, 'classifier__hidden_layer_sizes': (128,), 'classifier__batch_size': 64, 'classifier__alpha': 0.001, 'classifier__activation': 'tanh'}
12	MLP	0.888927	0.888419	0.888720	103.461112	0.110304	{'classifier__solver': 'sgd', 'classifier__learning_rate_init': 0.01, 'classifier__hidden_layer_sizes': (128, 64), 'classifier__batch_size': 64, 'classifier__alpha': 0.001, 'classifier__activation': 'tanh'}
13	MLP	0.888387	0.888387	NaN	49.605080	0.201961	{'classifier__solver': 'adam', 'classifier__learning_rate_init': 0.01, 'classifier__hidden_layer_sizes': (64,), 'classifier__batch_size': 64, 'classifier__alpha': 0.01, 'classifier__activation': 'relu'}
14	MLP	0.888796	0.888354	NaN	19.425479	0.202119	{'classifier__solver': 'adam', 'classifier__learning_rate_init': 0.001, 'classifier__hidden_layer_sizes': (64, 64), 'classifier__batch_size': 256, 'classifier__alpha': 0.0001, 'classifier__activation': 'relu'}
15	XGBoost	0.888526	0.888485	NaN	1.145410	0.226737	{'classifier__subsample': 1.0, 'classifier__reg_lambda': 1, 'classifier__reg_alpha': 10, 'classifier__n_estimators': 500, 'classifier__min_child_weight': 5, 'classifier__max_depth': 5, 'classifier__learning_rate': 0.1, 'classifier__gamma': 5, 'classifier__colsample_bytree': 1.0}
16	XGBoost	0.889353	0.888403	NaN	1.458273	0.237637	{'classifier__subsample': 0.8, 'classifier__reg_lambda': 10, 'classifier__reg_alpha': 0, 'classifier__n_estimators': 500, 'classifier__min_child_weight': 1, 'classifier__max_depth': 7, 'classifier__learning_rate': 0.2, 'classifier__gamma': 5, 'classifier__colsample_bytree': 1.0}
17	XGBoost	0.888698	0.888501	0.888621	0.651887	0.146361	{'classifier__subsample': 1.0, 'classifier__reg_lambda': 10, 'classifier__reg_alpha': 1, 'classifier__n_estimators': 100, 'classifier__min_child_weight': 1, 'classifier__max_depth': 5, 'classifier__learning_rate': 0.2, 'classifier__gamma': 5, 'classifier__colsample_bytree': 1.0}
18	XGBoost	0.888493	0.888403	NaN	1.147987	0.169854	{'classifier__subsample': 0.8, 'classifier__reg_lambda': 10, 'classifier__reg_alpha': 10, 'classifier__n_estimators': 300, 'classifier__min_child_weight': 1, 'classifier__max_depth': 7, 'classifier__learning_rate': 0.05, 'classifier__gamma': 5, 'classifier__colsample_bytree': 0.6}
19	XGBoost	0.888960	0.888485	NaN	0.991903	0.194132	{'classifier__subsample': 0.6, 'classifier__reg_lambda': 10, 'classifier__reg_alpha': 1, 'classifier__n_estimators': 100, 'classifier__min_child_weight': 5, 'classifier__max_depth': 7, 'classifier__learning_rate': 0.05, 'classifier__gamma': 1, 'classifier__colsample_bytree': 0.8}

Figure 4 - Best Hyper-Parameters



Model	Train_CV	Val_CV	Test_accuracy	Best_params
kNN	1.000000	0.888485	0.888523	{'classifier__weights': 'distance', 'classifier__p': 1, 'classifier__n_neighbors': 27, 'classifier__metric': 'minkowski', 'classifier__leaf_size': 40}
Logistic Regression	0.888387	0.888387	0.888425	{'classifier__solver': 'saga', 'classifier__penalty': 'l1', 'classifier__l1_ratio': 0.1, 'classifier__class_weight': None, 'classifier__C': np.float64(0.0001)}
MLP	0.888927	0.888419	0.888720	{'classifier__solver': 'sgd', 'classifier__learning_rate_init': 0.01, 'classifier__hidden_layer_sizes': (128, 64), 'classifier__batch_size': 64, 'classifier__alpha': np.float64(0.001), 'classifier__activation': 'tanh'}
XGBoost	0.888698	0.888501	0.888621	{'classifier__subsample': 1.0, 'classifier__reg_lambda': 10, 'classifier__reg_alpha': 1, 'classifier__n_estimators': 100, 'classifier__min_child_weight': 1, 'classifier__max_depth': 5, 'classifier__learning_rate': 0.2, 'classifier__gamma': 5, 'classifier__colsample_bytree': 1.0}

Figure 5 - Model Performance Comparison

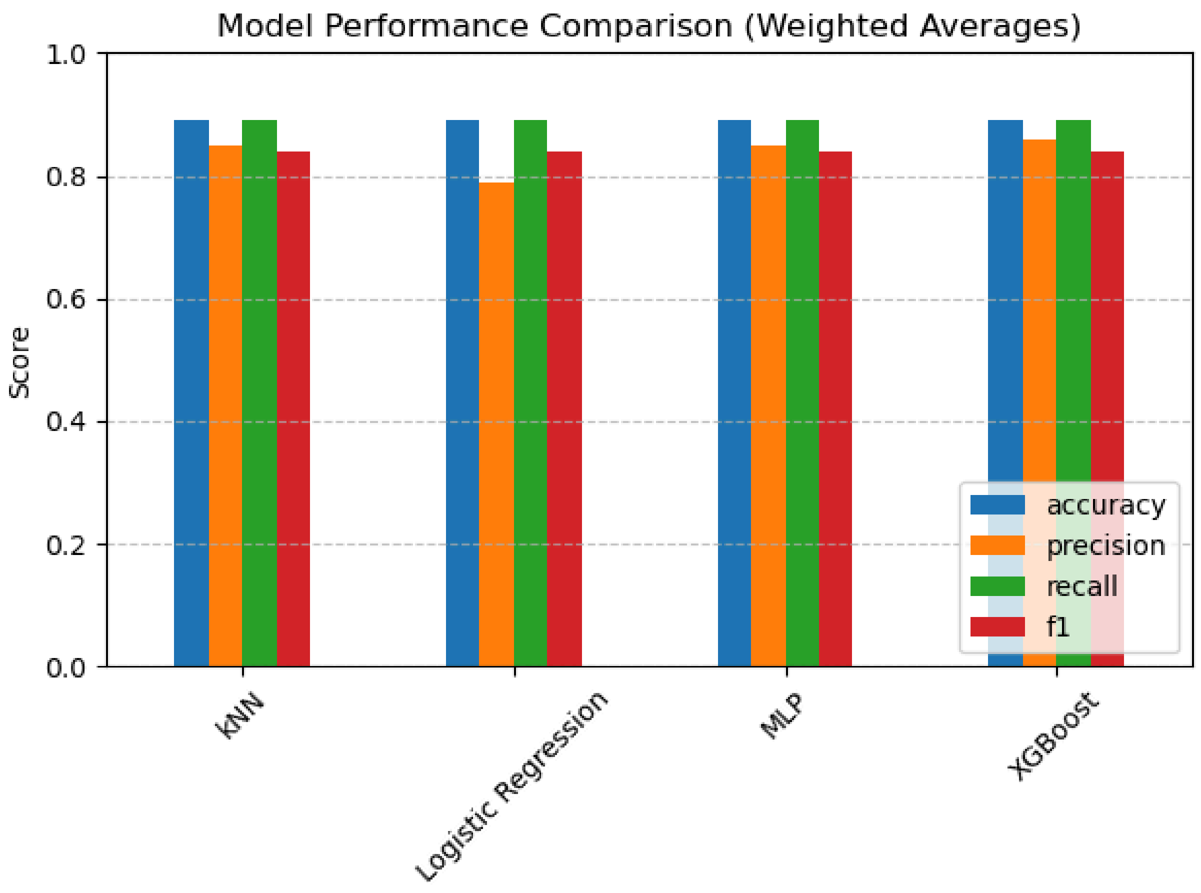


Figure 6 - Learning Curves

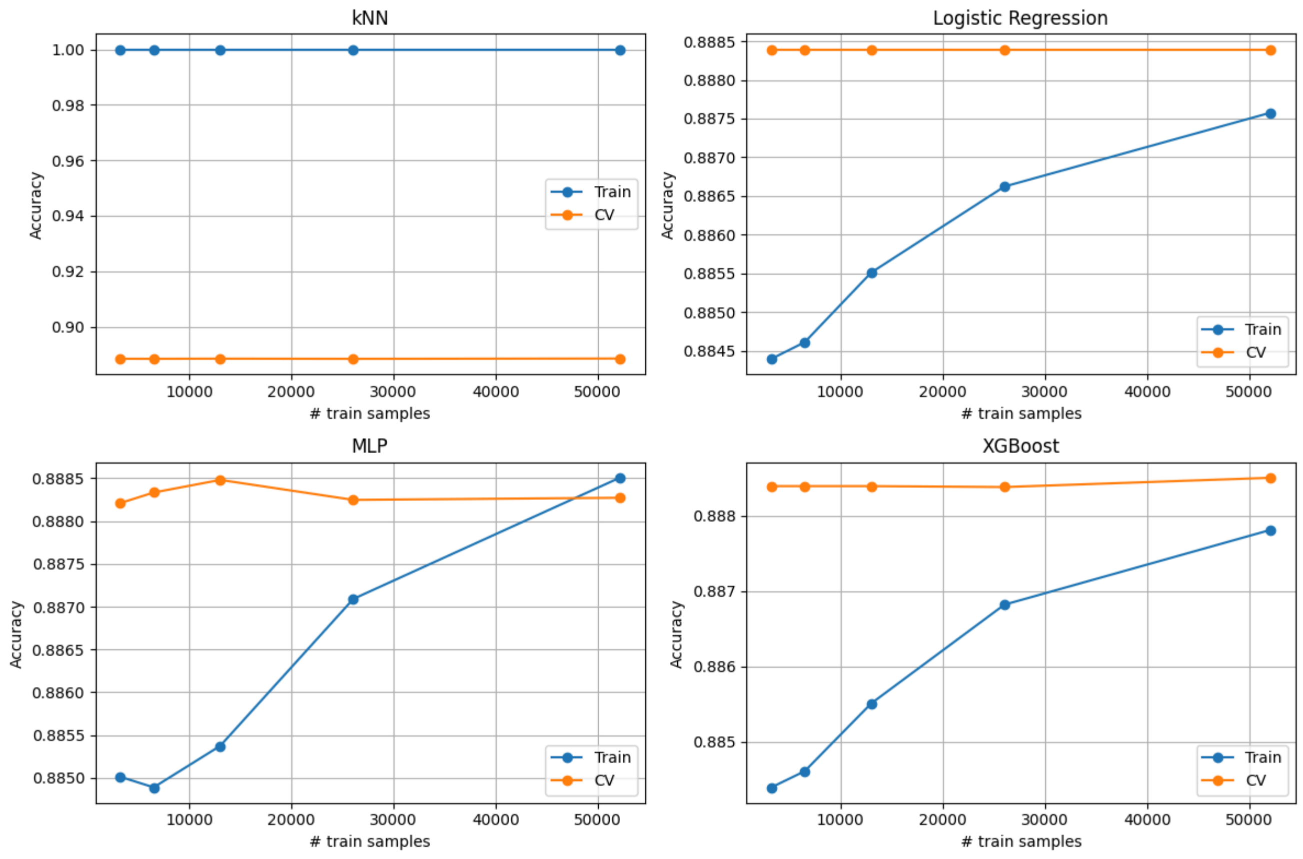
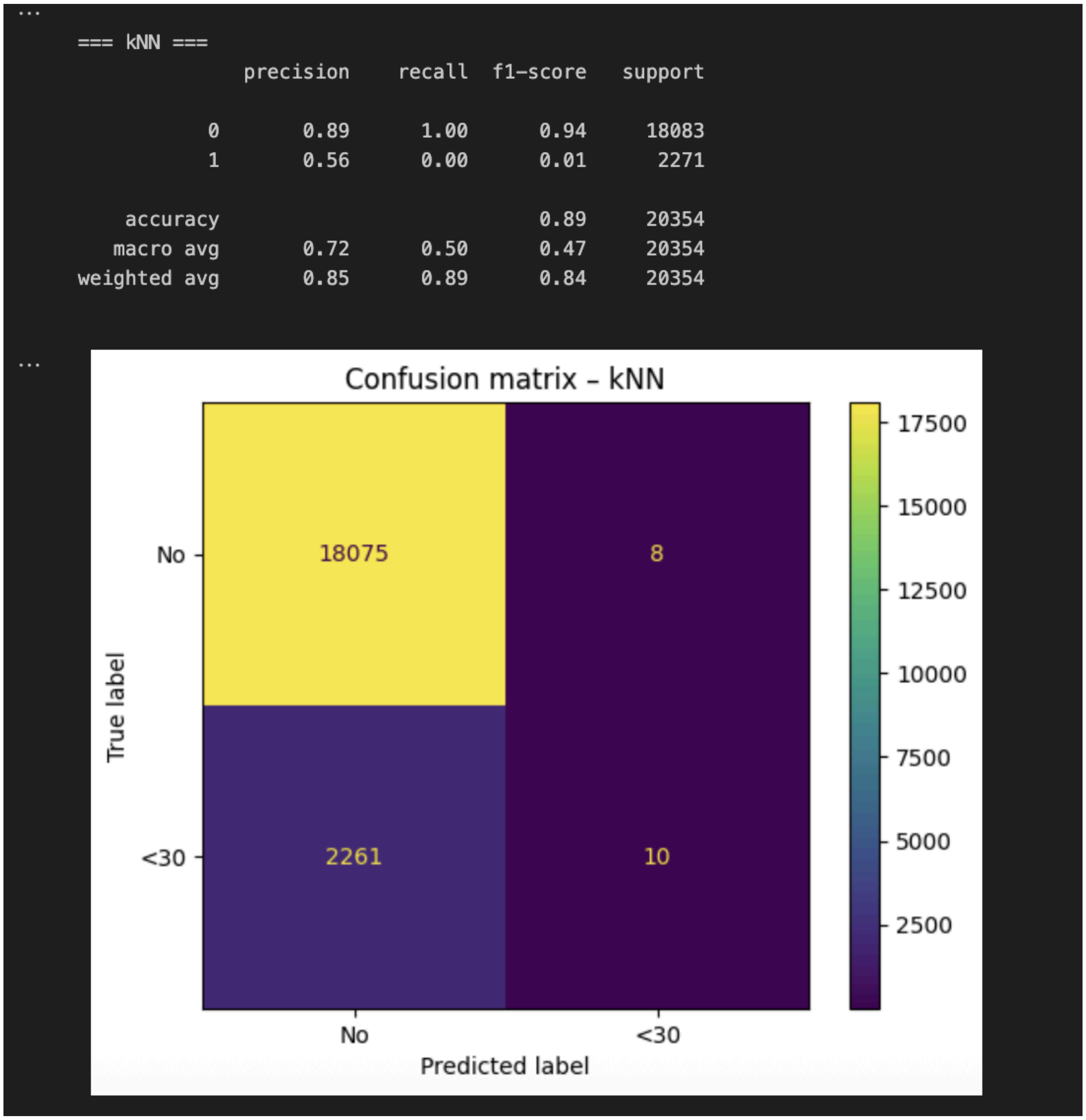


Figure 7 - Confusion Matrices

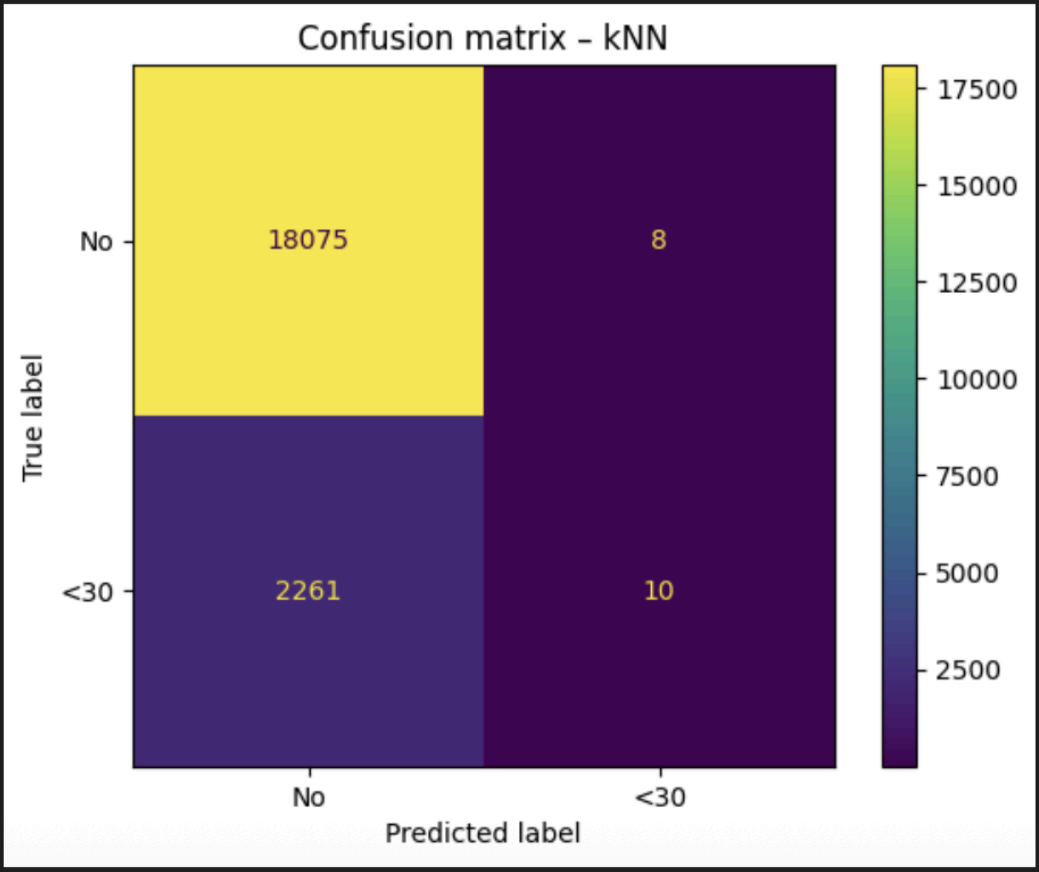


```
...
    === kNN ===
                  precision    recall  f1-score   support

               0       0.89      1.00      0.94     18083
               1       0.56      0.00      0.01      2271

        accuracy                           0.89     20354
       macro avg       0.72      0.50      0.47     20354
    weighted avg       0.85      0.89      0.84     20354

...
```

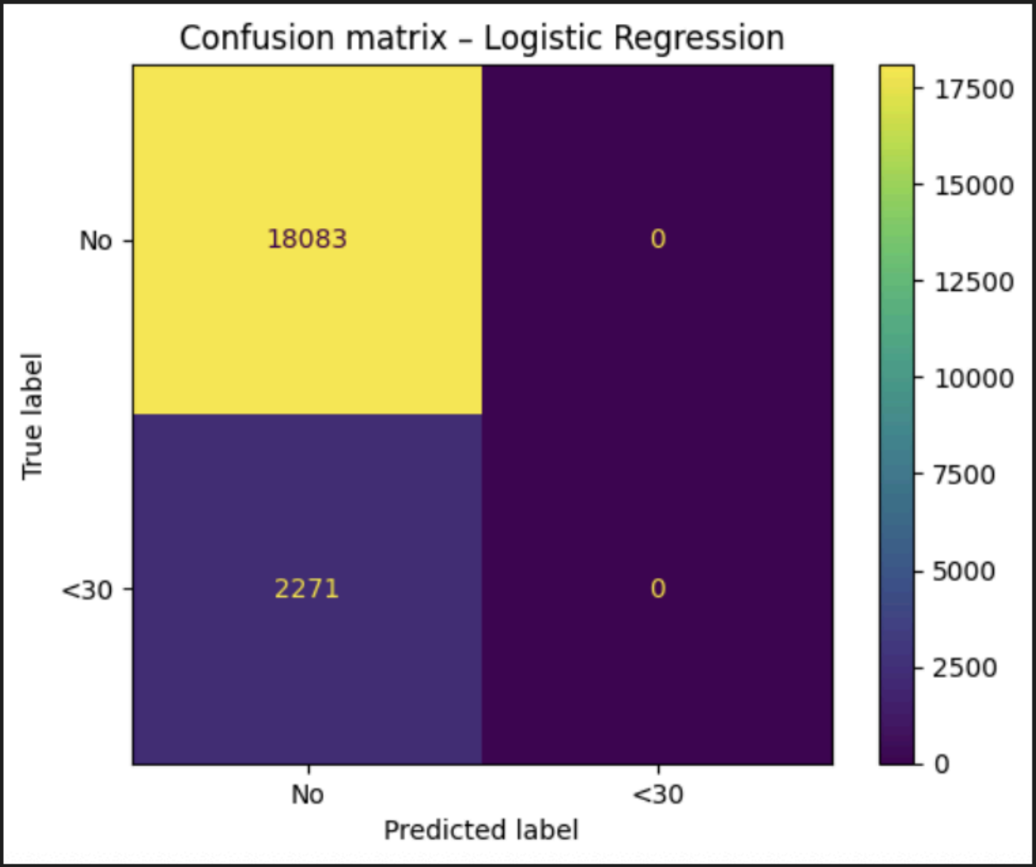


```
=== Logistic Regression ===
              precision    recall  f1-score   support

           0       0.89      1.00      0.94     18083
           1       0.00      0.00      0.00      2271

    accuracy                           0.89     20354
   macro avg       0.44      0.50      0.47     20354
weighted avg       0.79      0.89      0.84     20354

/opt/homebrew/Caskroom/miniconda/base/envs/cs178/lib/python3.12/site-packages/sklearn/me
  _warn_prf(average, modifier, f"{metric.capitalize()} is", result.shape[0])
/opt/homebrew/Caskroom/miniconda/base/envs/cs178/lib/python3.12/site-packages/sklearn/me
  _warn_prf(average, modifier, f"{metric.capitalize()} is", result.shape[0])
/opt/homebrew/Caskroom/miniconda/base/envs/cs178/lib/python3.12/site-packages/sklearn/me
  _warn_prf(average, modifier, f"{metric.capitalize()} is", result.shape[0])
```

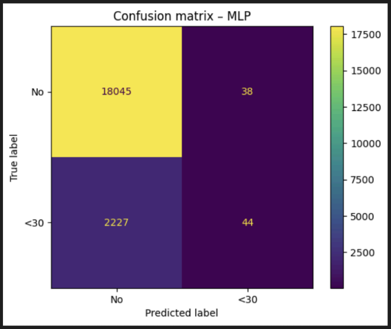


```
=== MLP ===
              precision    recall  f1-score   support

           0       0.89      1.00      0.94     18083
           1       0.54      0.02      0.04      2271

    accuracy                           0.89     20354
   macro avg       0.71      0.51      0.49     20354
weighted avg       0.85      0.89      0.84     20354

/opt/homebrew/Caskroom/miniconda/base/envs/cs178/lib/python3.12/site-packages/sklearn/utils/extmath.py:203: RuntimeWarning: divide by zero encountered in matmul
  ret = a @ b
/opt/homebrew/Caskroom/miniconda/base/envs/cs178/lib/python3.12/site-packages/sklearn/utils/extmath.py:203: RuntimeWarning: overflow encountered in matmul
  ret = a @ b
/opt/homebrew/Caskroom/miniconda/base/envs/cs178/lib/python3.12/site-packages/sklearn/utils/extmath.py:203: RuntimeWarning: invalid value encountered in matmul
  ret = a @ b
```



```
=== XGBoost ===
              precision    recall  f1-score   support

           0       0.89      1.00      0.94     18083
           1       0.64      0.00      0.01      2271

    accuracy                           0.89     20354
   macro avg       0.77      0.50      0.47     20354
weighted avg       0.86      0.89      0.84     20354
```

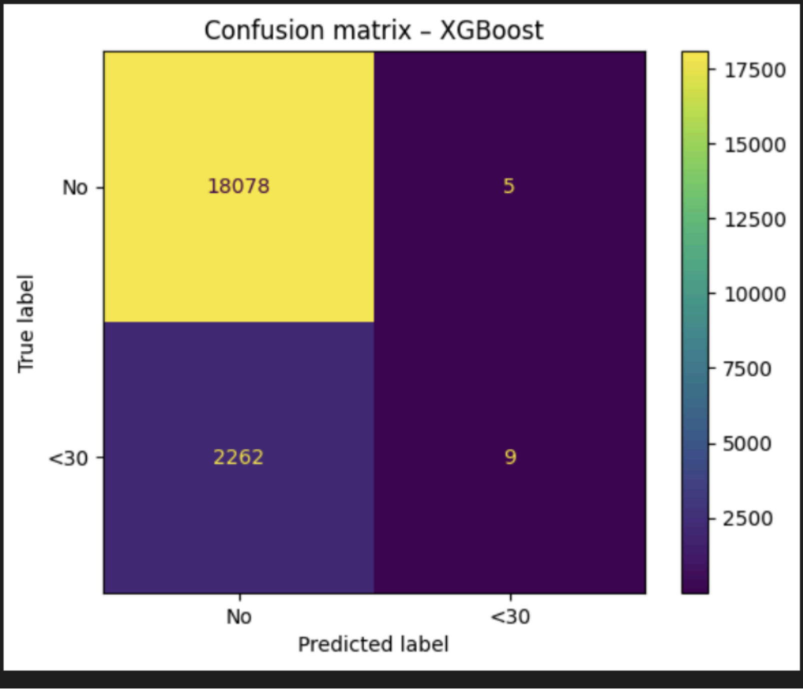


Figure 8 - Jaccard Similarity

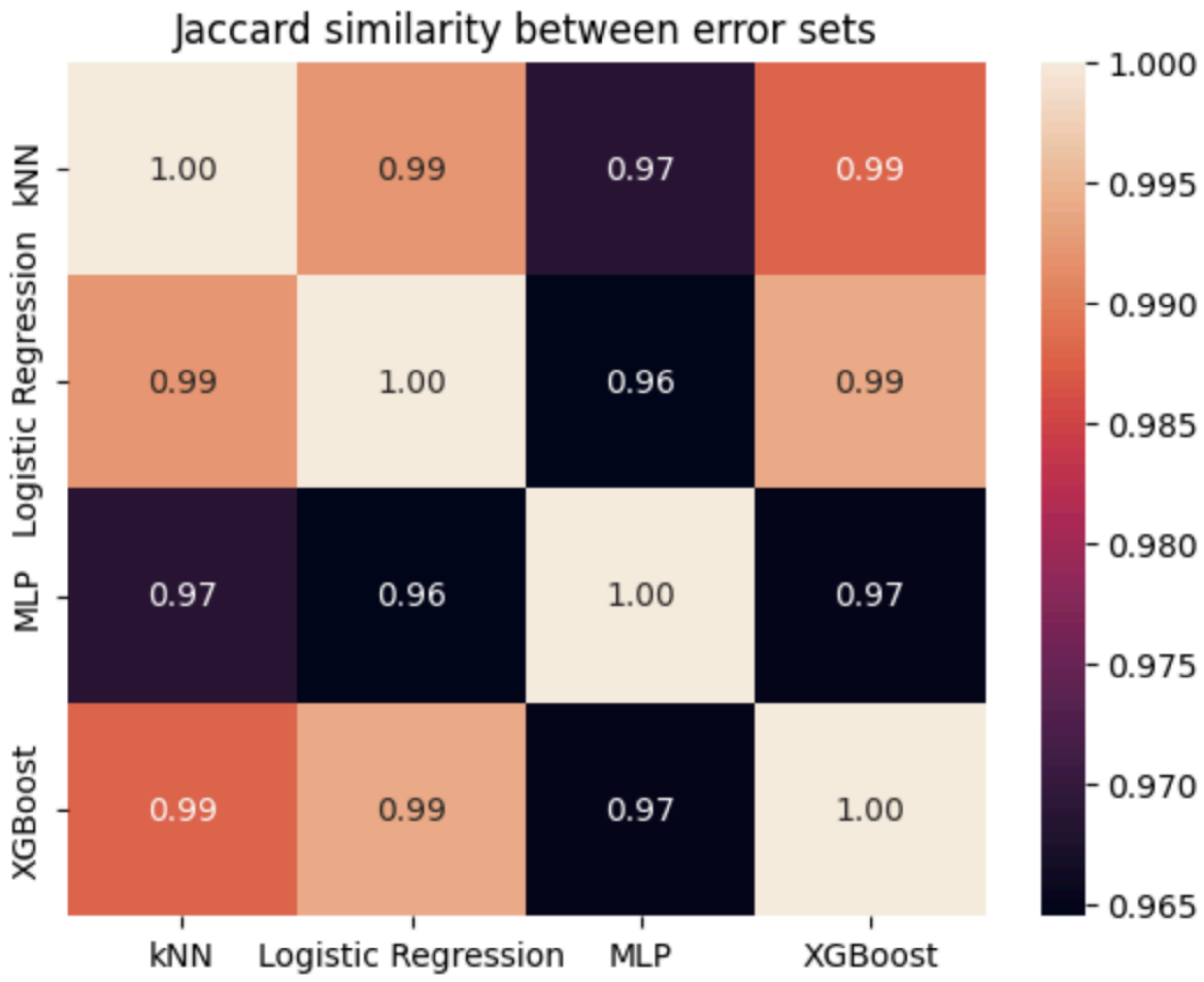
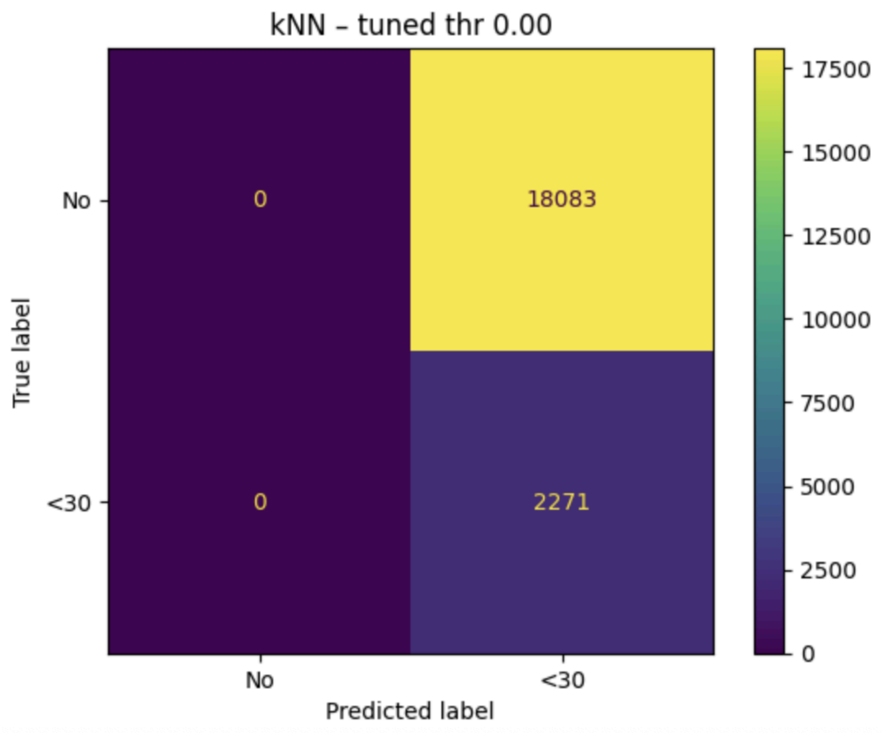
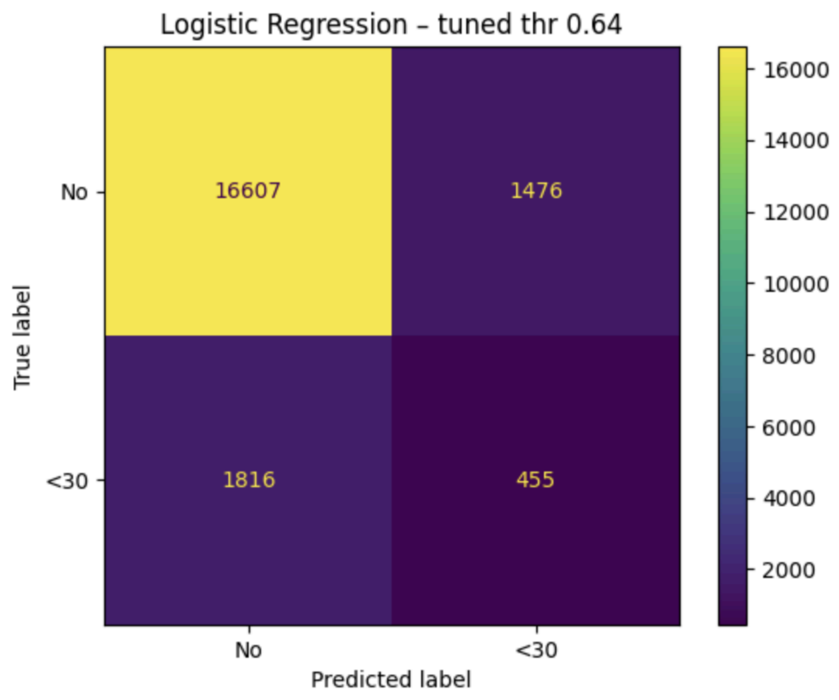
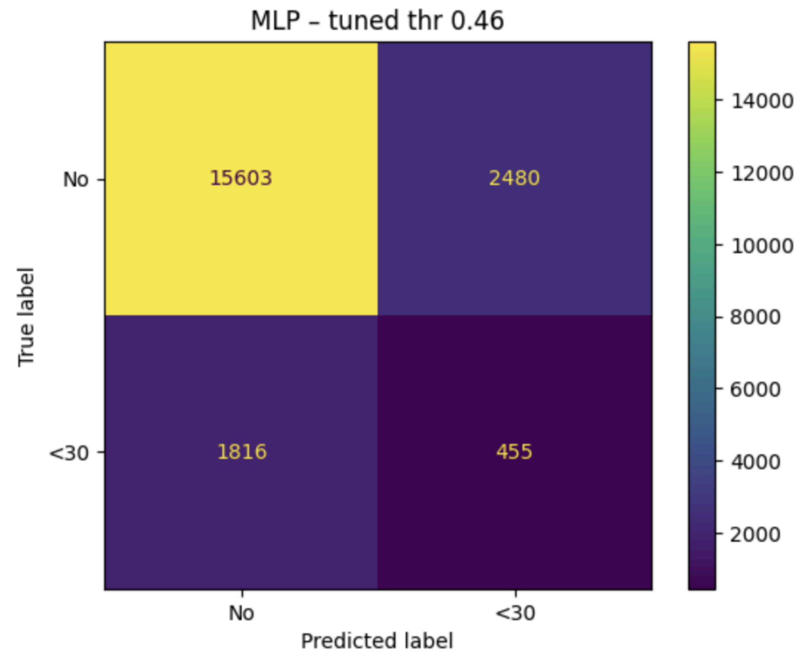
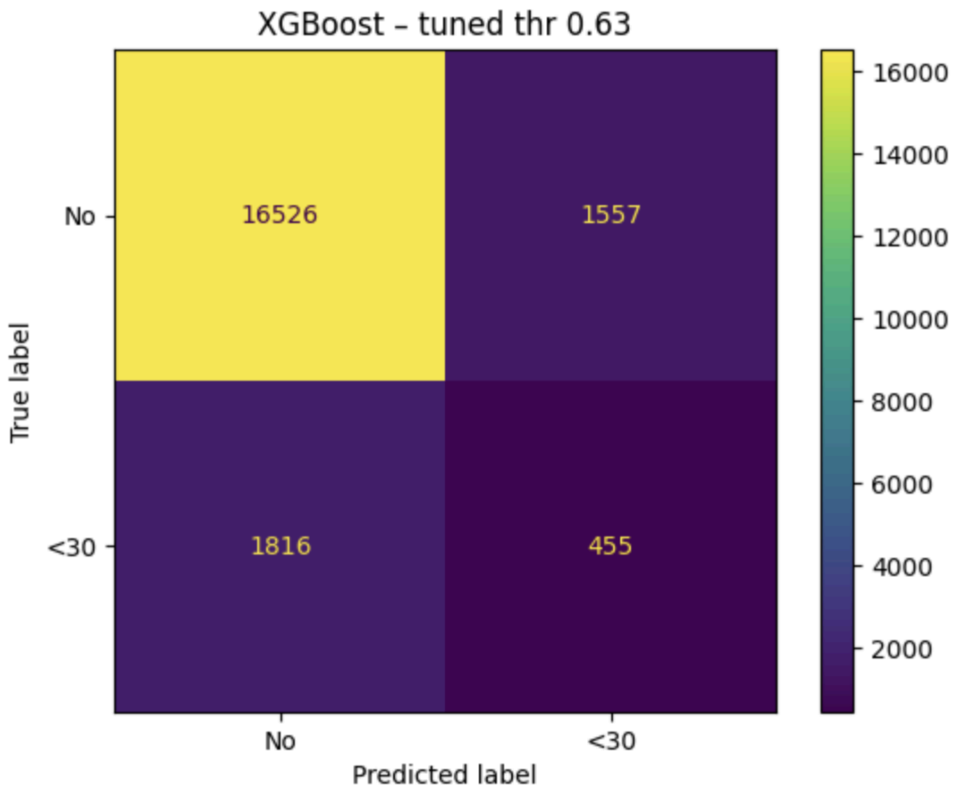


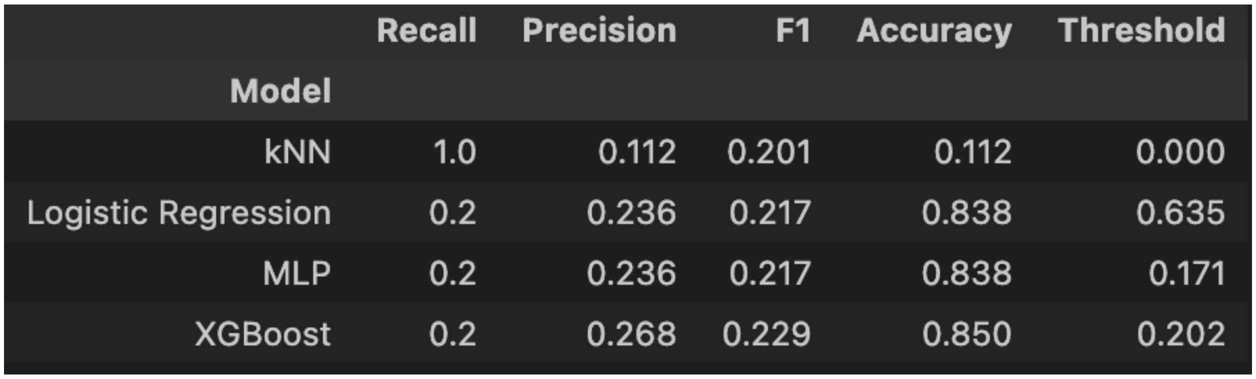
Figure 9 - New Models Optimized on Recall






MLP – tuned thr 0.46
True label
No
<30
Predicted label
15603
2480
1816
455
14000
12000
10000
8000
6000
4000
2000


XGBoost – tuned thr 0.63
True label
No
<30
Predicted label
16526
1557
1816
455
16000
14000
12000
10000
8000
6000
4000
2000



	Recall	Precision	F1	Accuracy	Threshold
Model					
kNN	1.0	0.112	0.201	0.112	0.000
Logistic Regression	0.2	0.236	0.217	0.838	0.635
MLP	0.2	0.236	0.217	0.838	0.171
XGBoost	0.2	0.268	0.229	0.850	0.202