Karam Hussain 27857
File name: kaggle_task2_final.ipynb

Task 2: Kaggle Submission

a) Loading dataset

Fetching and reading

```
train = pd.read_csv('train1.csv')
test  = pd.read_csv('test.csv')   # used to generate final submission

# quick shape
print("Train shape:", train.shape)
print("Test shape: ", test.shape)
```

✓ 4.9s

```
Train shape: (296209, 67)
Test shape:  (126948, 66)
```

b) Train - test split

Splitting the data into 75-25 train test set with stratification due to imbalanced class problem.

Train-test split

```
X = train.drop('target', axis=1)
y = train['target']

X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.25, stratify=y, random_state=42
)
print("X_train", X_train.shape, "X_val", X_val.shape)

print(f"Target distribution in train: {y_train.value_counts(normalize=True).to_dict()}")
print(f"Target distribution in val: {y_val.value_counts(normalize=True).to_dict()}")
```

✓ 0.9s

```
X_train (222156, 66) X_val (74053, 66)
Target distribution in train: {0: 0.9487342227983939, 1: 0.05126577720160608}
Target distribution in val: {0: 0.9487259125221126, 1: 0.05127408747788746}
```

c) Imputing

Handling missing values, separating columns on their datatypes; binary, categorical, numeric in order to filter further. Filtered the columns with the help of the names of columns. Also noted that the categorical columns were actually not literally ‘categorical’ but instead in number form i.e. labelled encoded, which is an important detail.

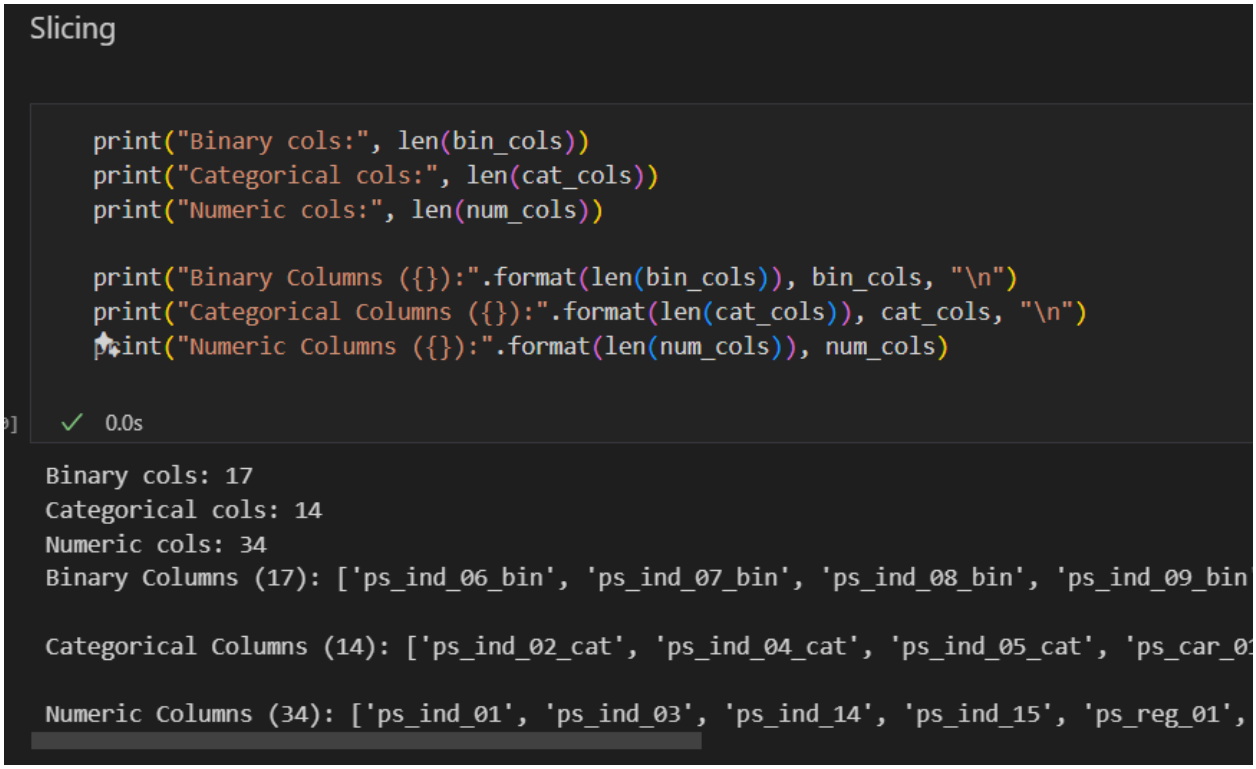
Used mode to fill in the missing values of the categorical columns and median to fill in the numerical columns, which will help in overcoming outliers if they will be present in the dataset and will be more robust.

Imputing

```
from sklearn.impute import SimpleImputer

bin_cols = [c for c in X_train.columns if '_bin' in c]
cat_cols = [c for c in X_train.columns if '_cat' in c]
num_cols = [c for c in X_train.columns if c not in bin_cols + cat_cols + ['id']]

# Categorical columns → mode (most_frequent)
imputer_cat = SimpleImputer(strategy='most_frequent')
X_train[cat_cols] = imputer_cat.fit_transform(X_train[cat_cols])
X_val[cat_cols] = imputer_cat.transform(X_val[cat_cols])

# Fill numeric NaNs with median from training data
median_vals = X_train[num_cols].median()
X_train[num_cols] = X_train[num_cols].fillna(median_vals)
X_val[num_cols] = X_val[num_cols].fillna(median_vals)

# fill binary columns with mode
imputer_bin = SimpleImputer(strategy='most_frequent')
X_train[bin_cols] = imputer_bin.fit_transform(X_train[bin_cols])
X_val[bin_cols] = imputer_bin.transform(X_val[bin_cols])
```

d) Slicing

Separating columns in terms of binary 17 , numeric 34 and categorical 14.



e) Preprocessors

Defining the preprocessors for all the intended columns to be used including the must evaluated ones.
Models:
-Categorical Naive Bayes, KNN, Decision Tree, Random Forest, Adaboost

Other models:
-Gaussian Naive Bayes, Catboost, XGBoost, LightGBM

For categorical naive bayes, we have decided to use categorical and binary columns to it, as the categorical columns are label encoded we must One-Hot encode them (14) so they can at least be taken as a bin_category rather than a magnitude.

For gaussian naive bayes, we will be passing numeric and binary columns to it, as for numeric we would have to scale them.

For KNN, which is a distance-based algorithm, we will pass it all the columns binary, numeric and categorical. Numeric would have to be scaled, categorical would be one-hot encoded to be treated as bin_cat rather than labelled as we don't want the category in ‘number’ be treated as a magnitude, that would cause wrong interpretation. And binary would go as it is.

Separate preprocessor for the tree models, which can be used by all of them where we will pass the default columns as it is, binary categorical and numeric , as they don't need scaling or further encoding.

Preprocessors

```python
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler


preprocessor_cnb = ColumnTransformer([
    ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False), cat_cols),
    ('bin', 'passthrough', bin_cols)
])


preprocessor_gnb = ColumnTransformer([
    ('num', StandardScaler(), num_cols),
    ('bin', 'passthrough', bin_cols)
])


preprocessor_knn = ColumnTransformer([
    ('num', StandardScaler(), num_cols),
    ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False), cat_cols),
    ('bin', 'passthrough', bin_cols)
])


# No scaling or encoding required
preprocessor_tree = ColumnTransformer([
    ('num', 'passthrough', num_cols),
    ('cat', 'passthrough', cat_cols),
    ('bin', 'passthrough', bin_cols)
])
```

f) Defining models

Defining all models with their respective preprocessors.
(all could not be captured in the screenshot)

```python
# ============================
# 🧠 DEFINE BASE MODELS (no hyperparameters yet)
# ============================

models = {
    "CategoricalNB": Pipeline([
        ("preprocessor", preprocessor_cnb),
        ("model", CategoricalNB())
    ]),

    "GaussianNB": Pipeline([
        ("preprocessor", preprocessor_gnb),
        ("model", GaussianNB())
    ]),

    "KNN": Pipeline([
        ("preprocessor", preprocessor_knn),
        ("model", KNeighborsClassifier())
    ]),

    "DecisionTree": Pipeline([
        ("preprocessor", preprocessor_tree),
        ("model", DecisionTreeClassifier(random_state=42))
    ]),

    "RandomForest": Pipeline([
        ("preprocessor", preprocessor_tree),
        ("model", RandomForestClassifier(random_state=42))
    ]),

    "AdaBoost": Pipeline([
        ("preprocessor", preprocessor_tree),
        ("model", AdaBoostClassifier(random_state=42))
    ]),

    "XGBoost": Pipeline([
        ("preprocessor", preprocessor_tree),
        ("model", xgb.XGBClassifier(
            random_state=42,
            use_label_encoder=False,
            eval_metric='logloss'
        ))
    ]),
```

g) Defining hyperparameter grid

Hyperparameters with multiple values.
(all of them could not be captured in the picture)

```
# ==========================
# 🎛️ DEFINE HYPERPARAMETER GRIDS (lightweight)
# ==========================

param_grids = {

    # --- Categorical Naive Bayes ---
    "CategoricalNB": {
        "model__alpha": [0.5, 1.0, 2.0]
    },

    # --- Gaussian Naive Bayes ---
    "GaussianNB": {
        "model__var_smoothing": [1e-9, 1e-7]
    },

    # --- K-Nearest Neighbors ---
    "KNN": {
        "model__n_neighbors": [5, 7, 9],
        "model__weights": ["uniform", "distance"]
    },

    # --- Decision Tree ---
    "DecisionTree": {
        "model__criterion": ["gini", "entropy"],
        "model__max_depth": [5, 7],
        "model__min_samples_split": [2, 5],
        "model__min_samples_leaf": [1,2]
    },
```

h) Random search CV

Due to the large number of hyp, we will first apply random search for speed efficiency and local optimum to narrow down the best ones with CV=3 for robustness and 5 iterations.

```
random_search = RandomizedSearchCV(
    estimator=pipe,
    param_distributions=param_grids[name],
    n_iter=5,                  # number of random combinations to try (tune as needed)
    scoring='roc_auc',
    cv=3,
    n_jobs=1,                  # use all cores for speed
    random_state=42,
    verbose=1
)
```

Applying the hyperparameters found randomly gave us a good estimate about the models without consuming too much time.

-KNN: got the last position with the slowest train time of 2110 s and slowest test time as well with 176 s, due to the fact that it first needs feature scaling then it needs to find the distance with each record and calculate it, lowest AUROC of 0.52.

-Decision Tree: did better than KNN with AUROC of 0.602, train time was approx 47 s, with the fastest test time of 0.15 s. As no scaling required and predicts fast due to already made rules.

-Categorical NB: did better than both decision tree and KNN, with auroc of 0.605, with a bit better train time of 43 s which is fast due to the fact it just calculates probabilities and later just need to calculate a new one using those thats why fast test time as well of 1.68 s.

-Random Forest: did better than the 3 above them with AUROC of 0.62 with comparatively higher train time as it trains a lot of trees together and fast test time of 1.03s, it decorrelates the tree.

-AdaBoost: did better than the 4 above, with 0.625 AUROC, double the training time of 604s and test time also of 2.7 s, uses sequential method of training trees with weights that's why took higher time for a large dataset.

-for the other models, Catboost won among all with highest AUROC of 0.636 with a higher training time of 210s but very fast prediction of 0.23s, where as xgboost and lightgbm were not so far behind and were 0.63+ as well in second and third respectively, stating that for large datasets these algorithms were superior compared to the base models. (xgboost and lightgbm had faster training time as well than catboost).
Another model used was gaussian naive bayes due to superiority of numeric columns but it got second last position, although much better than KNN with an AUROC of 0.598 and not so far behind than DT, it had the fastest train time among all and test time was also fast.
Overall winner was Catboost among the nine models.

	Model	Best AUROC	Train Time (s)	Predict Time (s)	Best Params
0	CatBoost	0.636402	209.99	0.23	{'model__learning_rate': 0.1, 'model__iteratio...
1	XGBoost	0.633440	56.28	0.25	{'model__subsample': 0.8, 'model__n_estimators...
2	LightGBM	0.633219	58.96	0.37	{'model__subsample': 1.0, 'model__num_leaves':...
3	AdaBoost	0.625238	604.79	2.72	{'model__n_estimators': 100, 'model__learning_...
4	RandomForest	0.622117	375.65	1.03	{'model__n_estimators': 100, 'model__min_sampl...
5	CategoricalNB	0.605720	43.53	1.68	{'model__alpha': 0.5}
6	DecisionTree	0.602350	47.06	0.15	{'model__min_samples_split': 5, 'model__min_sa...
7	GaussianNB	0.598768	9.57	0.49	{'model__var_smoothing': 1e-09}
8	KNN	0.521121	2110.85	176.62	{'model__weights': 'distance', 'model__n_neigh...

🏆 Best tuned model: CatBoost (AUROC = 0.6364)

i) Grid search CV

After narrowing down the top 3 models catboost, lightgbm and xgboost, we applied grid search with CV=3 to find the optimal set of hyp in order to maximize their AUROC, searching on a larger set of values this time.

```
# --- Narrow parameter grids ---
cat_param_grid = {
    'model__depth': [5, 6, 7],
    'model__learning_rate': [0.05, 0.1],
    'model__iterations': [300, 500],
    'model__l2_leaf_reg': [3, 5, 7]
}

xgb_param_grid = {
    'model__max_depth': [4, 5, 6],
    'model__learning_rate': [0.05, 0.1],
    'model__n_estimators': [200, 400],
    'model__subsample': [0.8, 1.0],
    'model__colsample_bytree': [0.8, 1.0]
}

lgbm_param_grid = {
    'model__num_leaves': [15, 31, 63],
    'model__learning_rate': [0.05, 0.1],
    'model__n_estimators': [200, 400],
    'model__subsample': [0.8, 1.0]
}
```

After tuning the hyp and applying, catboost was still number one , while a slight overtake from lightgbm to dethrone xgboost, all of them had increased similar AUROCs of 0.635+ . while catboost had the slowest train and test time, xgboost had the fastest test time and lightgbm provided a good balance with fastest train time and a moderate test time as well comparatively.

	Model	Best AUROC	Train Time (s)	Predict Time (s)	Best Params
0	CatBoost	0.636566	2945.27	0.57	{'model__depth': 6, 'model__iterations': 300, ...
1	LightGBM	0.635346	459.74	0.47	{'model__learning_rate': 0.05, 'model__n_estim...
2	XGBoost	0.635004	1186.04	0.31	{'model__colsample_bytree': 0.8, 'model__learn...

🥇 Best Fine-Tuned Model: CatBoost (AUROC = 0.636566)

j) Full train on data using these models with hyp + file creation

We will train these top 3 models using the hyperparameters from grid search to maximize AUROC, and create kaggle submission files.

```python
# --- Define the top 3 best parameter sets from your fine-tuned GridSearch ---
best_params_dict = {
    "CatBoost": {
        'iterations': 300,
        'depth': 6,
        'learning_rate': 0.05,
        'l2_leaf_reg': 3
    },
    "LightGBM": {
        'num_leaves': 15,
        'learning_rate': 0.05,
        'n_estimators': 200,
        'subsample': 0.8
    },
    "XGBoost": {
        'max_depth': 4,
        'learning_rate': 0.05,
        'n_estimators': 200,
        'subsample': 0.8,
        'colsample_bytree': 0.8
    }
}
```

k) Extending: Stacking ensemble of the top3 models + file creation

Using the method of stacking taught in class, on the top 3 best models catboost, lightgbm and xgboost with final classifier of logistic regression with 1000 iterations and CV=5 for robustness and better convergence of AUROC. This meta-model will help strengthen the score using the help of all 3 models.

The ensemble got trained fast in 260s and predicted as well in 1s with an increased AUROC of 0.6370.

Trained this model on full train set for file submission.

```
# --- Base models (using the tuned hyperparameters) ---
base_models = [
    ('catboost', CatBoostClassifier(verbose=0, random_state=42, **best_params_dict["CatBoost"])),
    ('lightgbm', lgb.LGBMClassifier(random_state=42, **best_params_dict["LightGBM"])),
    ('xgboost', xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42, **best_params_dict["XGBoost"]))
]

# --- Meta-model (simple and strong for stacking) ---
meta_model = LogisticRegression(max_iter=1000, random_state=42)

# --- Build the stacking ensemble ---
stacking_model = StackingClassifier(
    estimators=base_models,
    final_estimator=meta_model,
    cv=5,               # 5-fold stacking
    stack_method='predict_proba',
    n_jobs=1
)
```

```
✅ Ensemble trained in 260.37s
✅ Validation predictions complete in 1.04s

📊 Validation AUROC for Stacking Ensemble: 0.637070
```

l) Fine tuning Ensemble (final part)

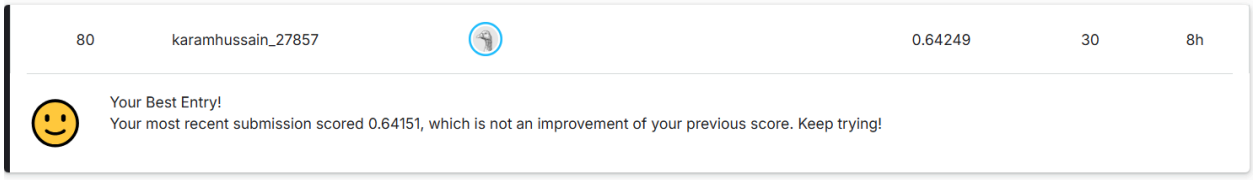
Decided to experiment further on fine tuning the meta-model ensemble with a small set of hyp with two/three values and CV=2 to minimize computation time and also getting the optimal set from these using grid search.

The meta model got tuned in 13,176 s with the following hyperparameters in the second pic below, slight increase with highest AUROC of 0.6371 and 1.2s test time, then trained this on full set as well for submission.

```python
# --- Meta-model hyperparameter grid (Logistic Regression) ---
meta_param_grid = {
    'final_estimator__C': [0.01, 0.1, 1],
    'final_estimator__solver': ['lbfgs', 'saga'],
    'final_estimator__max_iter': [1000, 1500],
    'final_estimator__class_weight': [None, 'balanced'],
    'final_estimator__penalty': ['l2','l1']

}

# --- Define stacking classifier with placeholder meta-model ---
stack_base = StackingClassifier(
    estimators=estimators,
    final_estimator=LogisticRegression(random_state=42),
    stack_method='predict_proba',
    n_jobs=1
)

# --- Grid search to tune Logistic Regression meta-model ---
print("\n🔍 Tuning meta-model (Logistic Regression) inside stacking ensemble...")
start_tune = time.time()
grid_meta = GridSearchCV(
    estimator=stack_base,
    param_grid=meta_param_grid,
    scoring='roc_auc',
    cv=2,
    verbose=2,
    n_jobs=1
)
```

```
✅ Meta-model tuning complete in 13176.39s

🏆 Best meta-model hyperparameters:
   final_estimator__C: 0.1
   final_estimator__class_weight: None
   final_estimator__max_iter: 1000
   final_estimator__penalty: l1
   final_estimator__solver: saga

🎯 Validation AUROC (stacked ensemble): 0.63710
⏱️ Prediction time: 1.20s
```

m) Kaggle final leaderboard score



Highest kaggle score of 0.64249

(table analysis on the next page)

****Final AUROC Comparison table****

With Randomized search HYP on base models for narrowing down:

	Method	AUROC
2	CatBoost	0.6364
3	XGBoost	0.6334
4	LightGBM	0.6332
5	AdaBoost	0.6252
6	RandomForest	0.6221
7	CategoricalNB	0.6057
8	DecisionTree	0.6023
9	GaussianNB	0.5987
10	KNN	0.5211

With Grid search HYP on top 3 models and Ensemble:

Method	AUROC
Tuned Stacking Ensemble	0.6371
Stacking Ensemble	0.637
CatBoost	0.6365
LightGBM	0.6353
XGBoost	0.635

(insights on next page)

Analysis:
Tree-based models consistently outperformed distance and probability-based algorithms. CatBoost achieved the highest single-model AUROC (0.6365), leveraging ordered boosting and efficient handling of categorical features. The stacking ensemble showed strong complementary learning among gradient-boosting models, with its non-tuned version scoring slightly higher on Kaggle's public leaderboard (AUROC = 0.64158) compared to its tuned variant (0.6371 on validation). This suggests that light regularization or less aggressive tuning allowed better generalization to the unseen test data. Simpler learners such as KNN and Naive Bayes performed poorly due to high dimensionality, feature scaling sensitivity, and limited ability to capture complex feature interactions. Overall, ensemble tree methods proved the most robust and generalizable choice for this dataset.

The close performance among CatBoost, XGBoost, and LightGBM suggests all boosting algorithms reached a similar bias–variance balance, though CatBoost's ordered boosting and categorical handling likely gave it a slight edge. Interestingly, the non-tuned stacking ensemble scored higher on Kaggle, implying that lighter regularization can generalize better to unseen data than aggressively optimized hyperparameters. Overall, the results reaffirm that gradient-boosted ensembles are the most reliable approach for structured tabular data, providing both stability and strong predictive power.

(hyperparameter insight next page)

CatBoost: iterations = 300, depth = 6, learning_rate = 0.05, l2_leaf_reg = 3

LightGBM: num_leaves = 15, learning_rate = 0.05, n_estimators = 200, subsample = 0.8

XGBoost: max_depth = 4, learning_rate = 0.05, n_estimators = 200, subsample = 0.8, colsample_bytree = 0.8

Stacking (meta-model): C = 0.1, penalty = 'l1', solver = 'saga', max_iter = 1000, class_weight = None

Hyperparameter Insights:
The tuned boosting models consistently favored moderate learning rates (0.05) and shallow to medium tree depths, emphasizing gradual learning and controlled model complexity. Both LightGBM and XGBoost used subsampling (0.8) to inject randomness and reduce overfitting, while CatBoost's depth of 6 with mild L2 regularization achieved a balanced bias–variance trade-off. The stacking ensemble's meta-model (Logistic Regression with L1 penalty, C = 0.1) further enhanced generalization by performing implicit feature selection and assigning higher weights to the most reliable base learners. Overall, the chosen parameters highlight that moderate regularization and conservative learning were key to achieving stable AUROC performance across models.