EEE 485: Statistical Learning and Data Analytics

Final Report

Predictive Modeling for Startup Valuation and Success

Mehreen Irfan 22201004
Şebnem Sarı 21901816

1. Introduction

This report summarises the work carried out and the conclusions drawn from the development of a predictive modelling system for startup valuation and success. It outlines the progress made during the initial stages of the project as well as the changes introduced after receiving peer feedback. The report also explains the reasoning behind key design decisions and the steps that ultimately led to the final results.

The main aim of this project is to build a machine learning model that can analyse startup-related data and generate predictions for important business indicators. The model is designed as a black-box system that takes objective input features and produces estimates for a startup's valuation and potential exit outcome.

As the project progressed, several pivots were made based on model performance and feedback from peers. These included changes to feature/preditor selection, preprocessing methods, and modelling approaches in order to better capture real-world startup behaviour. This iterative process was an important part of improving both the reliability and practicality of the final system.

2. Dataset

The dataset used in this study contains 500 startup entries and was chosen to be representative of real startup scenarios while still being manageable for experimentation. The columns of the data set are as follows [1]:

- Region (categorical)
- Year Founded (continuous)
- Industry (categorical)
- Funding Rounds (continuous)
- Funding Amount (M USD, continuous)
- Number of Employees (continuous)
- Revenue (M USD, continuous)
- Valuation (M USD, continuous)
- Market Share (% continuous)
- Profitable (binary)
- Exit Status (categorical)

3. Preliminary Work [2]

3.1. Initial Dataset and Preprocessing

The project began with the analysis of a dataset containing 500 startup entries, which was selected to mimic real-world scenarios while remaining manageable for experimental iterations. The feature set encompassed continuous metrics such as Funding Amount, Revenue, and Number of Employees, alongside categorical variables including Region, Industry, and Exit Status. To align the raw data with realistic magnitudes, specific parameters were scaled during this initial phase; specifically, Valuation was scaled by a factor of 0.2, Revenue by 0.5, and Employees by 0.3. A critical component of the preliminary pipeline was the data preprocessing strategy, which involved Z-standardization ($z = \frac{x-\mu}{\sigma}$) for continuous columns and integer mapping for categorical features. In this approach, each category was assigned a single integer digit to process the data for the prediction models. As detailed in later sections, this integer mapping strategy was subsequently identified as a limitation that introduced unintended ordinal relationships, negatively impacting model performance.

3.2. Regression Model Development

To establish performance baselines, three linear regression models were implemented entirely from scratch using Python's NumPy library.

Linear Regression (Ordinary Least Squares - OLS):

The first model, Ordinary Least Squares (OLS), calculated regression coefficients using the closed-form solution $\beta = (X^T X)^{-1} X^T y$ and utilized the pseudo-inverse to handle potential matrix singularity.

Ridge Regression:

The second model, Ridge Regression, utilized a modified closed-form solution $\beta = (X^T X - \lambda I)^{-1} X^T y$ with L2 regularization to penalize large coefficients and manage multicollinearity.

Lasso Regression:

The third model, Lasso Regression, was developed using an iterative coordinate descent algorithm with conditional updates to enforce sparsity and feature selection.

To evaluate these models, a custom validation framework was developed using Generalized K-Fold Cross-Validation with K=5, ensuring reproducible splits via fixed random seeding.

3.3. Classification Model Development

For the classification task of predicting "Exit Status," a Multi-Layer Perceptron (MLP) was constructed robustly using a native NumPy implementation. The architecture consisted of an input layer, one hidden layer, and an output layer, utilizing the Sigmoid activation function to produce output probabilities. The training process proceeded iteratively using the Backpropagation algorithm and Gradient Descent to minimize loss over a set limit of 5000 epochs.

3.4. Initial Results and Limitations

The preliminary results highlighted significant challenges in modeling the non-linear startup ecosystem with linear tools. For the Valuation target, the Lasso model achieved the best performance, yielding an R^2 of 0.6376 and a Test RMSE of 145.06 M USD. However, the models failed to capture any meaningful

signal for Market Share prediction, with the best-performing Ridge model yielding a negative R^2 of -0.0623, indicating performance worse than a simple mean baseline. The MLP classifier yielded an accuracy of approximately 33%, effectively performing at the level of random guessing. These findings confirmed that the dataset contained high noise and non-linear patterns that simple linear models and integer-mapped features could not effectively capture, necessitating the strategic pivots detailed in the following sections.

4. Work Done

4.1. Identified Limitations

Based on the feedback received during the preliminary work evaluation, as well as insights gained through experimentation and iterative trial-and-error analysis, several fundamental issues were identified. These issues were rooted partly in the codebase, and partly, rather to a great extent in the structure and limitations of the dataset itself. Addressing these challenges led to multiple pivots and strategic adjustments, which are also explained in greater detail in the relevant sections below.

Firstly and most importantly, the dataset exhibited strong non-linear characteristics and weak linear correlations between the input features and the targets. This resulted in weak accuracy and unreliable regression as well as classification performance, preventing the linear models from reliably fulfilling the intended objectives of the system. This was further reinforced by quantising these weak correlations as shown in the matrix below:

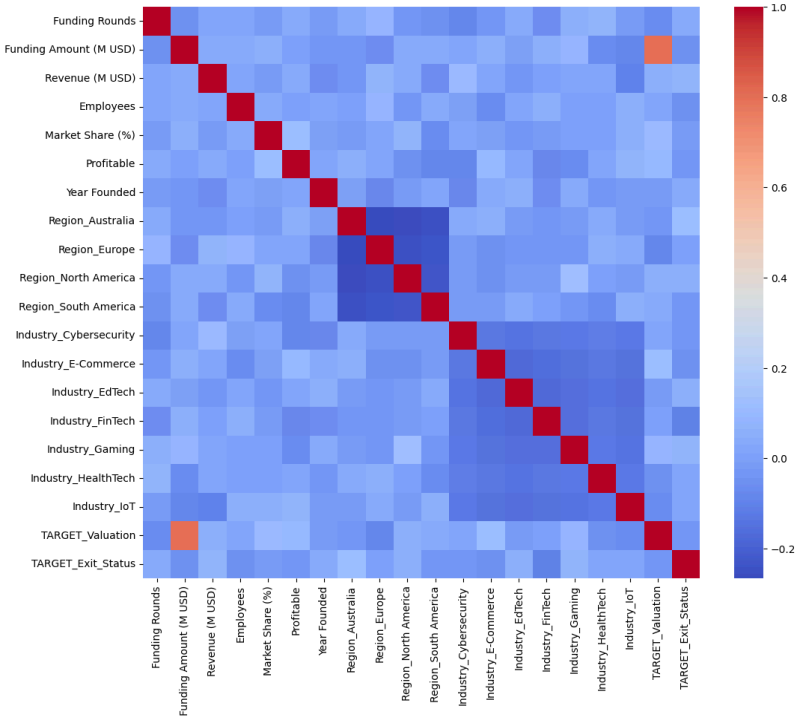

Fig.4.1. Dataset Correlation Matrix

This emerged as a root cause of the failure of the initially implemented linear classifier (MLP), as well as the accuracy being even worse than random selection in predicting one of the previously selected target variables - Market Share (%).

More so, in the case of Market Share(%), as reinforced by the correlation matrix as well as upon further non-technical research it was concluded that since it is a relative quantity depending mainly on the total market size and output [3] - features not available in the data set, the prediction was done more or less randomly, therefore resulting in sub-par results as shown below.

```
OLS  : R2 = -0.0685 | RMSE = 2.86
Ridge: R2 = -0.0684 | RMSE = 2.86
Lasso: R2 =  0.0044 | RMSE = 2.76

[Best Model Predictions: Market Share]
idx |   Actual    | Predicted
  0 |       0.21 |       4.98
  1 |       0.86 |       5.03
  2 |       4.11 |       4.92
  3 |       1.26 |       4.95
  4 |       3.85 |       4.91
 95 |       3.87 |       5.22
 96 |       2.23 |       4.94
 97 |       5.21 |       5.13
 98 |       7.75 |       5.24
 99 |       2.28 |       5.03
```

Fig.4.2. Market Share (%) Obtained Results

While in case of MLP, another critical reason why it did perform well lied in the heart of data preprocessing pipeline - the categorial features were mapped as integers, giving more weightage to some and less to other features, tampering mostly, but not only the MLP classifier but also the regressors.

4.2. Data Preprocessing

The data processing pipeline underwent a significant change in categorial features mapping. As opposed to the previously implemented integer mapping, One-hot encoding was implemented to these features instead including **Region** and **Industry**; the the feature set was appended by the addition of separate columns for every member of both of these features, only the column of corresponding category of every data instances was given the value of 1, the rest were kept as zero. This mitigated the weightage issue.

based on the observed performance of Market Share (%) as a target variable, it was instead kept as a feature. Although its individual correlation with the targets was relatively low, including it increased the dimensionality of the dataset and, given the limited sample size, provided an additional weak but useful correlator that contributed to more stable predictions across both regression and classification models [4].

The rest of the data preprocessing remained the same as that done in the preliminary work - manual resampling to mimic real-world setting, Z-standardisation of continuous features, and creation of feature set and the target vectors.

4.3. Regressors

Three distinct regression models were implemented entirely from scratch using Python's NumPy library. They were judged based on their RMSE and R^2 values.

4.3.1. Linear Regressor (Ordinary Least Squares - OLS)

The basic formulation of OLS was kept the same as done in the preliminary report with one minor change - addition of an epsilon value regularising term as given below (update equation) in order to prevent the algorithm from diverging in case of similar features interception.

$$\beta = (X^T X + \epsilon I)^{-1} X^T y$$

4.3.2. Ridge Regressor

The basic formulation of the Ridge Regressor was kept the same as done in the preliminary work. However, the training and evaluation of the model was made robust by implementing the iterative parameter tuning of the ridge penalty - lambda. The lambda giving the least RMSE was chosen (less weightage was given to R^2 since RMSE is a better metric for model selection [5]. The graph of Ridge regressor's performance as per RMSE for different values of lambda was obtained as follows:

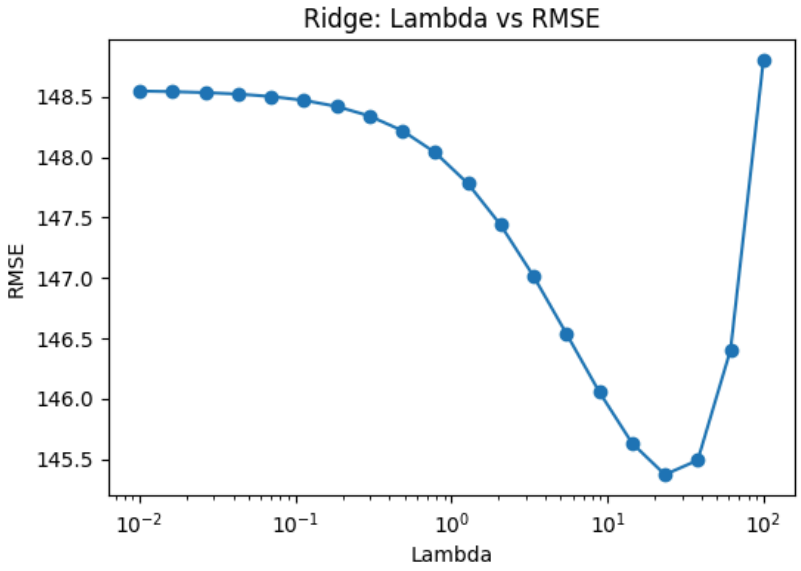

Fig.4.3. Ridge Regressor - RMSE vs λ

4.3.3. Lasso Regressor

The implementation logic of Lasso Regressor went through a major change - utilisation of gradient descent instead of coordinate descent as implemented in the preliminary work. Gradient Descent is generally better for imbalanced datasets because it naturally supports mini-batch resampling, which effectively prevents the model from ignoring the minority class during training [6]. The Gradient descent updates its weights in the new code-base as follows:

$$\widehat{w} = w(t) - \eta \,.\, 1/m \,.\, X^T (X\, w(t) + b - y)$$

$$w(t+1) = sigm\,(\widehat{w}) \,.\, \max(\,|\,w\,|\, - (\eta \,.\, \alpha), 0)$$

Moreso, similar to the Ridge Regressor, the training and evaluation of the model was made robust by implementing the iterative parameter tuning of the lasso penalty - lambda - as well as the learning rate since it greatly affects the gradient descent update as given above. The lambda and LR giving the least RMSE was chosen. The graph of Lasso regressor's performance as per RMSE for different values of lambda and LR was obtained as follows:

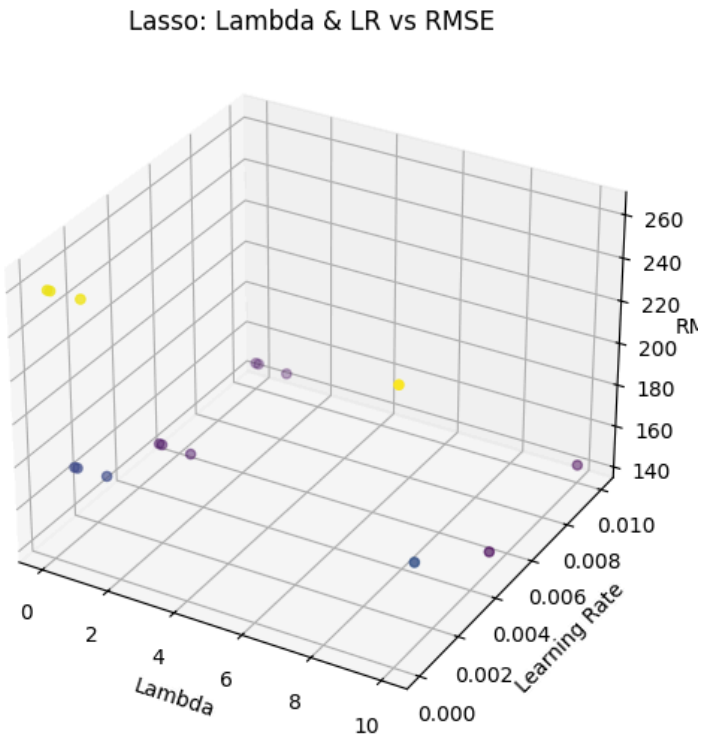

Fig.4.4. Ridge Regressor - RMSE vs λ and LR

4.4. Classifiers

Three distinct new classification models were implemented entirely from scratch using Python's NumPy library. The previously implemented MLP classifier was removed since its linear classification nature clashed with the objective of this project.

The evaluation metrics were also changed for this code; Kappa score [7], Macro F1-Score - the average of F1 scores for all classes, treating minority and majority classes equally, and class accuracies were used in place of global accuracy for the reason to be explained in later sections.

4.4.1. XGBoost Classifier

Ensemble-based tree methods have been shown to outperform linear classifiers in low-sample, weak-correlation settings due to their ability to capture higher-order feature interactions [8]. Therefore, two of the new implemented classifiers were kept ensemble based. In the implemented approach, gradient boosting constructs an additive model of weak learners by minimising a differentiable loss function. The weight updates in the code follow the given set of mathematical logic:

$$g_i = w_k(y_i - p_k(x_i)) \quad where\ g\ is\ the\ gradient\ of\ each\ tree$$

$$w_k(t+1) = w(t) + \eta\, h_t(x) \quad where\ h_t(x)\ is\ the\ tree\ prediction\ (leastsquares\ projection\ of\ g\ on\ the\ tree)$$

4.4.2. AdaBoost

AdaBoost or adaptive boost is another ensemble based classifier. It trains a sequence of weak learners by combining the results from individual single layer decision trees (stumps). After each round, it increases the sample weights of misclassified points, forcing the next stump to focus specifically on the hard-to-predict examples. This method is especially used due to its emphasis on minority class classification. The weight updates in the code follow the given set of mathematical logic:

$$\alpha_t = 1/2\, ln\,((1 - \epsilon_t)/\epsilon_t)$$

$$w_i(t+1) = w_i(t)\,.\, e^{-\alpha_t y_i \widehat{y_i}} \quad where\ \widehat{y_i}\ is\ the\ predicted\ label$$

4.4.3. Gaussian Naive Bayes

Gaussian Naive Bayes was chosen because it is a high-bias, low-variance model, which makes it very stable for small datasets like the one used in this project. Additionally, because it estimates the statistical distribution of each class independently, it remains sensitive to the minority class rather than being overwhelmed by the majority class frequency [9]. To make a prediction, it simply measures which class's statistical "shape" the new startup fits into best; it predicts the class with the highest log likelihood score as follows:

$$S_c = log\,(Prior_c - 1/2 \sum_j (log\,(2\pi\sigma_{c,j}^2) + ((x_j - \mu_{c,j})^2/\sigma_{c,j}^2)$$

$$where\ \sigma_{c,j}^2\ is\ the\ variance\ and\ \mu_{c,j}\ is\ the\ mean$$

5. Results

5.1. Regressors

As foretold in the previous section, the best regressor (one to be implemented) was chosen as follows:

The best individual regressors were obtained as follows:

```
 REGRESSION EVALUATION
OLS RMSE: 148.5537
Best Ridge Lambda: 23.3572, RMSE: 145.3710
Best Lasso Lambda: 10.0, LR: 0.01, RMSE: 143.4506
```

Fig.5.1. Best Individual Regressor Metrics

The best fit of these regressors was as follows:

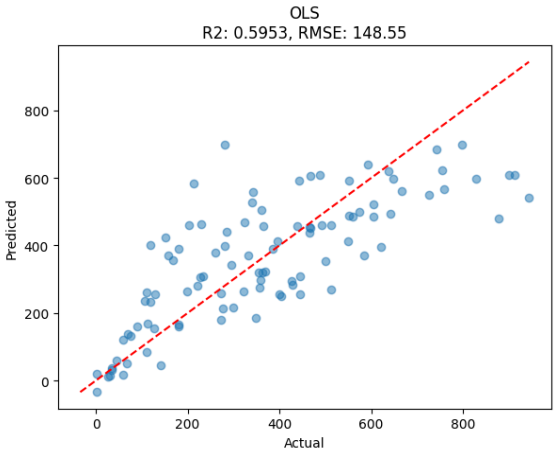

Fig.5.2.a. OLS Linear Fit

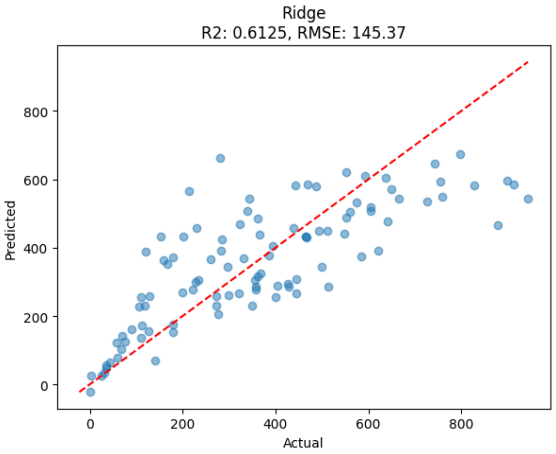

Fig.5.2.b. Ridge Linear Fit

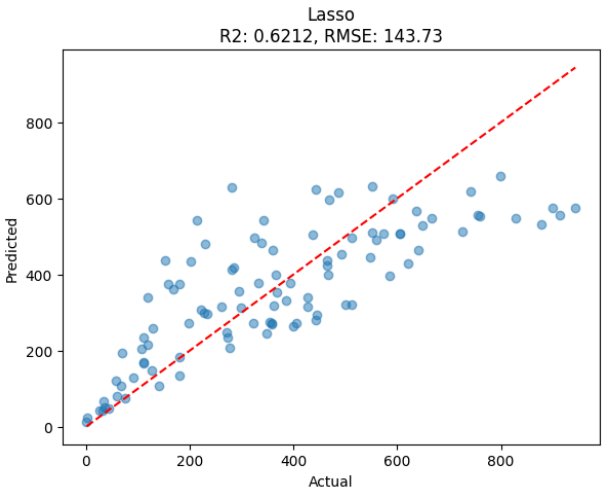

Fig.5.2.c. Lasso Linear Fit

As seen from the metrics scores, with less distinction between all three regressors, Lasso - with the least RMSE was chosen as the best regressor. Despite being a small dataset, all three models produced nearly identical results because the relatively high ratio of samples to features allowed the standard OLS model to be naturally stable, almost making the stabilization effects of regularization mostly unnecessary. While Lasso achieved a slight edge by successfully zeroing out irrelevant noise features, the convergence of all three scores confirms that the underlying data follows a robust linear pattern that all three algorithms captured somewhat effectively.

5.2. Classifiers

The three classifier models were judged and the best one was selected based on their Kappa score, F1-score, and class accuracy. These metrics were chosen over the global accuracy because, global accuracy is often a misleading indicator of performance when applied to imbalanced datasets like the oe employed. In the context of exit status, where private exits make up 75% of the entire data set while IPOs barely make up 7% (seen later in the confusion matrices), our models achieved over 70% accuracy simply by predicting private exits for every company. This "Accuracy Paradox" conceals the model's inability to detect the minority cases that are most valuable to the analysis.

To counter this paradox, inherent in imbalanced datasets, we prioritized class accuracy to ensure the model explicitly identifies rare exits (classes) rather than simply predicting the majority class. This was complemented by the F1 Score, which balances precision and sensitivity by heavily penalizing false positives and false negatives, ensuring a robust assessment of quality beyond simple correctness. Finally, Cohen's Kappa served as a statistical control to adjust for random chance, confirming that the model's performance stems from learning actual feature patterns rather than exploiting the dataset's skewed frequency distributions.

This resulted in the following metrics for each of the regressors and their respective confusion matrices showing their per class performances.

```
Training Gaussian NB...
Accuracy : 0.6600
Macro F1        : 0.3962
Kappa Score     : 0.0965
Class Accuracies: [0.22 0.14 0.81]
```

Fig.5.3.a. GNB Metrics

```
Training AdaBoost...
Accuracy : 0.7400
Macro F1        : 0.2852
Kappa Score     : 0.0054
Class Accuracies: [0.   0.   0.99]
```

Fig.5.3.b. AdaBoost Metrics

```
Training XGBoost...
Accuracy : 0.7500
Macro F1        : 0.2857
Kappa Score     : 0.0000
Class Accuracies: [0. 0. 1.]
```

Fig.5.3.c. XGBoost Metrics

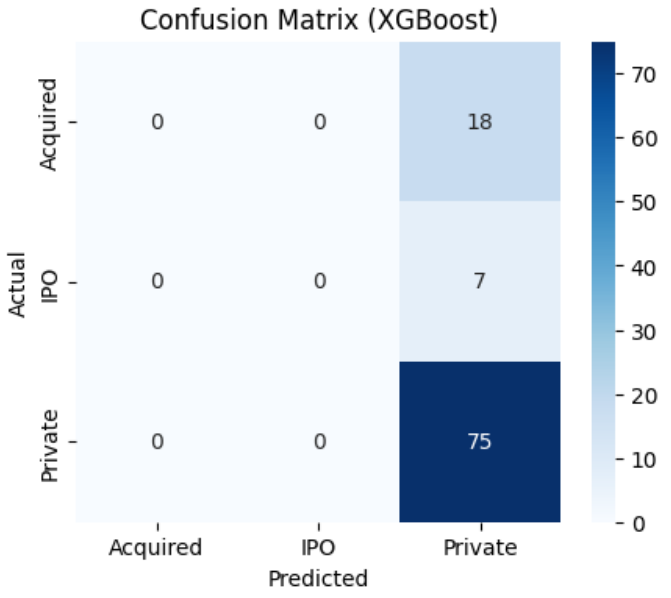

Fig.5.4.a. XGBoost Confusion Matrix

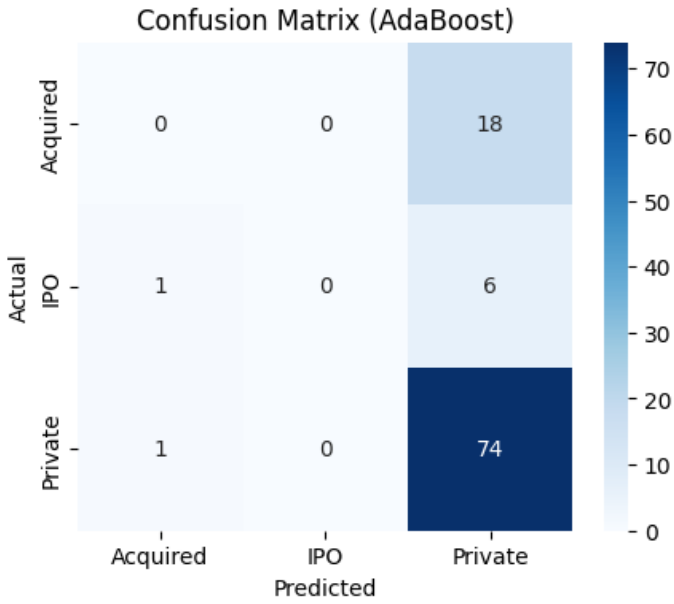

Fig.5.4.b. AdaBoost Confusion Matrix

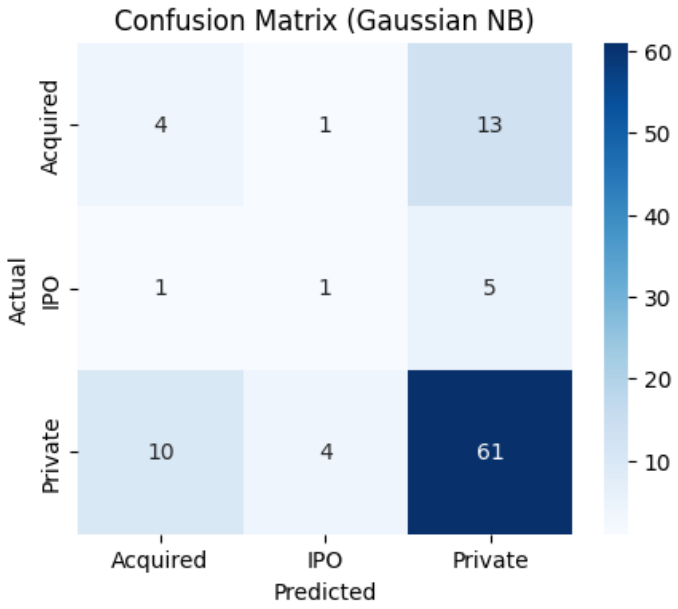

Fig.5.4.c. GNB Confusion Matrix

It can be seen from, both, the class accuracies as well as the confusion matrices, that despite having greater global accuracies, both of the ensemble classifiers label all three labels as private; Private exit status having 75% of weightage directly translates it to the model accuracy.

Gaussian Naive Bayes, however, seems to be giving greater weightage to minority classes as well, as seen from the class accuracy. The results are still not as accurate as required but it remains the best option because it successfully identified the rare successful startups that other models completely ignored. Unlike other algorithms methods that just memorized the majority failures, GNB used simple statistics to find the true patterns of high-value exits. This is the reason why GNB was selected.

Using the best chosen regressor and classifier models, the valuation and exit status new data points can be predicted in the MLOps pipeline created as follows:

```
DEMO FOR: QuantumHealth
Predicted Valuation      : $541.95 Million
Predicted Exit Status    : Private
```

Fig.5.5. Demo Prediction

6. Conclusion

The project initially aimed to utilize standard linear models and an MLP classifier on a 500-entry dataset to predict Valuation, Market Share, and Exit Status. However, preliminary results demonstrated that simple linear approaches and integer mapping for categorical features were insufficient for capturing the non-linear and noisy nature of startup ecosystems.

A significant portion of the work involved diagnosing these failures, leading to strategic pivots in both preprocessing and model selection. The transition from integer mapping to One-Hot Encoding eliminated unintended ordinal relationships that were hindering model performance. Furthermore, the decision to reclassify "Market Share" from a target variable to an input feature proved essential; while it was difficult to predict due to missing market-size data, it provided valuable dimensionality that stabilized the remaining predictions.

In the regression domain, the implementation of robust, scratch-built OLS, Ridge, and Lasso models revealed that the data possessed underlying linear patterns that could be effectively captured once noise was managed. While all three regressors showed convergence, the Lasso Regressor achieved the best performance with an RMSE of 143.45 M USD, successfully utilizing feature selection to filter out irrelevant signals.

For classification, the project moved away from the linear limitations of the MLP to explore ensemble methods and probabilistic models. While XGBoost and AdaBoost achieved high global accuracy, they failed to capture minority outcomes. Gaussian Naive Baye**s** emerged as the superior choice; despite lower global accuracy, its high-bias stability allowed it to effectively identify rare, high-value exit scenarios that other models ignored. Ultimately, this project demonstrated that while startup success is volatile, prioritizing class-specific sensitivity over global accuracy is essential for extracting reliable predictive signals from imbalanced data.

7. References

[1] S. Ashar, “Startup Growth and Funding Trends,” Kaggle, 2023. [Online]. Available: https://www.kaggle.com/datasets/samayashar/startup-growth-and-funding-trends.

[2] M. Irfan and Ş. Sarı, “Predictive Modeling for Startup Valuation and Success,” EEE 485: Statistical Learning and Data Analytics, First Report, Bilkent University, 2025. [Online]. Available: https://docs.google.com/document/d/1KwFnlnPn4uc2301hwad6u5pD6bMZTBDsRSa9PZTJ8U8.

[3] D. Carlton and J. Perloff, *Modern Industrial Organization*, 4th ed. Boston, MA, USA: Pearson/Addison Wesley, 2005.

[4] Y. Li, H. G. Hong, S. E. Ahmed, and Y. Li, “Weak signals in high-dimensional regression: Detection, estimation and prediction,” *Applied Stochastic Models in Business and Industry*, vol. 35, no. 2, pp. 283–298, 2019. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/31666801/

[5] “RMSE vs R-squared,” *Data Science Stack Exchange*, Stack Exchange, 2018. [Online]. Available: https://datascience.stackexchange.com/questions/100605/rmse-vs-r-squared.

[6] Serrano.Academy, "Coordinate Descent Vs Gradient Descent?," *YouTube*, Dec. 28, 2018. [Online Video]. Available: https://www.youtube.com/watch?v=lC1fNSo2JKc.

[7] J. Mohanan, "How to use Cohen's Kappa Statistic for ML Model verification," *Medium*, May 02, 2024. [Online]. Available: https://medium.com/@jayamohanmohanan/how-to-use-cohens-kappa-statistic-for-ml-model-verification-6c66258b4ae9.

[8] J. H. Friedman, “Greedy function approximation: A gradient boosting machine,” *Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, 2001.

[9] S. Ray, "6 Easy Steps to Learn Naive Bayes Algorithm with codes in Python and R," *Analytics Vidhya*, Sep. 11, 2017. [Online]. Available: https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/.

Appendix A - Loading + Preprocessing Code

```
### M. Irfan, S. Sari

## loading and preprocessing
data = pd.read_csv('startup_data.csv')
data["Startup Name"] = data["Startup Name"].str.extract(r'(\d+)')

data["Valuation (M USD)"] *= 0.25
data["Revenue (M USD)"] *= 0.5
data["Employees"] = data["Employees"] * 0.3

#display(data.head())

def preprocess_pipeline(df, is_training=True, stats=None):
    df_n = df.copy()

    cont = ["Funding Amount (M USD)", "Employees", "Revenue (M USD)", "Year Founded", "Market
Share (%)"]
    cat = ["Region", "Industry"]

    for c in cont + cat:
        if c not in df_n.columns: df_n[c] = 0

    if is_training:
        means = df_n[cont].mean()
        stds = df_n[cont].std().replace(0, 1)
        stats = {'means': means, 'stds': stds}
    else:
        means, stds = stats['means'], stats['stds']

    df_n[cont] = (df_n[cont] - means) / stds

    for c in cat: df_n[c] = df_n[c].astype(str)
    df_n = pd.get_dummies(df_n, columns=cat, drop_first=True)

    if is_training:
        features = df_n.drop(["Startup Name", "Valuation (M USD)", "Exit Status"], axis=1,
errors='ignore').columns.tolist()
        stats['features'] = features
    else:
        features = stats['features']
        for col in features:
            if col not in df_n.columns: df_n[col] = 0
        df_n = df_n[features]

    return df_n, stats

processed_df, saved_stats = preprocess_pipeline(data, is_training=True)

X_all = processed_df[saved_stats['features']].values.astype(float)
```

```
y_val = processed_df["Valuation (M USD)"].values

labels = sorted(data["Exit Status"].unique())
exit_map = {k: v for v, k in enumerate(labels)}
inv_exit_map = {v: k for k, v in exit_map.items()}
y_exit = processed_df["Exit Status"].map(exit_map).values.astype(int)
```

Appendix B - Correlation Maps

```
finaldf = processed_df[saved_stats['features']].copy()
finaldf['TARGET_Valuation'] = y_val
finaldf['TARGET_Exit_Status'] = y_exit

plt.figure(figsize=(12, 10))
corr_matrix = finaldf.corr()
sns.heatmap(corr_matrix, cmap='coolwarm', annot=False)
plt.title("Dataset Correlation Matrix")
plt.show()
```

Appendix C - Helper Functions

```
def r2_score(y_true, y_pred):
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    if ss_tot == 0: return 0.0
    return 1 - (ss_res / ss_tot)

def rmse(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred)**2))

def accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred)

def train_test_split(X, y, test_size=0.2):
    np.random.seed(42)
    indices = np.arange(X.shape[0])
    np.random.shuffle(indices)
    split_idx = int(X.shape[0] * (1 - test_size))
    return X[indices[:split_idx]], X[indices[split_idx:]], y[indices[:split_idx]], y[indices[split_idx:]]

def confusion_matrix(y_true, y_pred, n_classes):
    cm = np.zeros((n_classes, n_classes), dtype=int)
    for t, p in zip(y_true, y_pred):
        cm[int(t), int(p)] += 1
    return cm

def precision_per_class(cm):
    precisions = []
    for i in range(len(cm)):
```

```
        tp = cm[i, i]
        fp = np.sum(cm[:, i]) - tp
        if tp + fp == 0: precisions.append(0.0)
        else: precisions.append(tp / (tp + fp))
    return np.array(precisions)

def recall_per_class(cm):
    recalls = []
    for i in range(len(cm)):
        tp = cm[i, i]
        fn = np.sum(cm[i, :]) - tp
        if tp + fn == 0: recalls.append(0.0)
        else: recalls.append(tp / (tp + fn))
    return np.array(recalls)

def f1_per_class(precision, recall):
    f1 = []
    for p, r in zip(precision, recall):
        if p + r == 0: f1.append(0.0)
        else: f1.append(2 * p * r / (p + r))
    return np.array(f1)

def kappa_score(y_true, y_pred, n_classes):
    cm = confusion_matrix(y_true, y_pred, n_classes)
    n_samples = np.sum(cm)
    po = np.trace(cm) / n_samples
    pe = 0
    for i in range(n_classes):
        row_sum = np.sum(cm[i, :])
        col_sum = np.sum(cm[:, i])
        pe += (row_sum * col_sum)
    pe /= (n_samples * n_samples)
    if pe == 1: return 1.0
    return (po - pe) / (1 - pe)

def macro_average(metric_array):
    return np.mean(metric_array)
```

Appendix D - Regressors

```
class LinearRegression:
    def fit(self, X, y):
        X_b = np.c_[np.ones((X.shape[0], 1)), X]
        I = np.eye(X_b.shape[1]) * 1e-6
        self.theta = np.linalg.pinv(X_b.T @ X_b + I) @ X_b.T @ y
    def predict(self, X):
        X_b = np.c_[np.ones((X.shape[0], 1)), X]
        return X_b.dot(self.theta)

class RidgeRegression:
    def __init__(self, lambda_=1.0):
```

```
      self.lambda_ = lambda_
    def fit(self, X, y):
        X_b = np.c_[np.ones((X.shape[0], 1)), X]
        I = np.eye(X_b.shape[1]); I[0, 0] = 0
        self.theta = np.linalg.inv(X_b.T @ X_b + self.lambda_ * I) @ X_b.T @ y
    def predict(self, X):
        X_b = np.c_[np.ones((X.shape[0], 1)), X]
        return X_b.dot(self.theta)

class LassoRegression:
    def __init__(self, lambda_=0.1, iterations=1000, learning_rate=0.01):
        self.lambda_, self.iter, self.lr = lambda_, iterations, learning_rate
    def _soft_threshold(self, x, lambda_param):
        return np.sign(x) * np.maximum(np.abs(x) - lambda_param, 0)
    def fit(self, X, y):
        m, n = X.shape
        self.w = np.zeros(n); self.b = 0
        for _ in range(self.iter):
            pred = X.dot(self.w) + self.b
            residuals = pred - y
            dw_mse = (1/m) * X.T.dot(residuals)
            db = (1/m) * np.sum(residuals)
            w_temp = self.w - self.lr * dw_mse
            self.b -= self.lr * db
            self.w = self._soft_threshold(w_temp, self.lambda_ * self.lr)
    def predict(self, X):
        return X.dot(self.w) + self.b
```

Appendix E - Classifiers

```
class TreeEstimator:
    def __init__(self, max_depth=5):
        self.max_depth = max_depth; self.tree = None
    def fit(self, X, y): self.tree = self._grow(X, y)
    def predict(self, X): return np.array([self._traverse(x, self.tree) for x in X])
    def _grow(self, X, y, depth=0):
        if len(y) == 0: return 0
        if depth >= self.max_depth or len(np.unique(y)) == 1 or len(X) < 5:
            return np.bincount(y.astype(int)).argmax()
        n_features = X.shape[1]
        feat_idxs = np.random.choice(n_features, int(np.sqrt(n_features)), replace=False)
        best_feat, best_thresh = self._best_split(X, y, feat_idxs)
        if best_feat is None: return np.bincount(y.astype(int)).argmax()
        left_idx = X[:, best_feat] <= best_thresh
        right_idx = X[:, best_feat] > best_thresh
        if np.sum(left_idx) == 0 or np.sum(right_idx) == 0: return np.bincount(y.astype(int)).argmax()
        return {"f": best_feat, "t": best_thresh, "l": self._grow(X[left_idx], y[left_idx], depth + 1), "r":
self._grow(X[right_idx], y[right_idx], depth + 1)}
    def _best_split(self, X, y, feats):
        best_gain = -1; split_feat, split_thresh = None, None
        for f in feats:
```

```
            thresholds = np.unique(X[:, f])
            if len(thresholds) > 10: thresholds = np.percentile(thresholds, [25, 50, 75])
            for t in thresholds:
                gain = self._gini_gain(y, X[:, f], t)
                if gain > best_gain: best_gain = gain; split_feat = f; split_thresh = t
        return split_feat, split_thresh
    def _gini_gain(self, y, col, thresh):
        parent_gini = self._gini(y)
        left = y[col <= thresh]; right = y[col > thresh]
        if len(left) == 0 or len(right) == 0: return 0
        return parent_gini - ((len(left) / len(y)) * self._gini(left) + (len(right) / len(y)) * self._gini(right))
    def _gini(self, y):
        probs = np.bincount(y.astype(int)) / len(y)
        return 1 - np.sum(probs ** 2)
    def _traverse(self, x, node):
        if not isinstance(node, dict): return node
        if x[node["f"]] <= node["t"]:
            return self._traverse(x, node["l"])
        else:
            return self._traverse(x, node["r"])

class XGBoost:
    def __init__(self, n_estimators=40, learning_rate=0.1, max_depth=4):
        self.n_estimators = n_estimators; self.lr = learning_rate; self.max_depth = max_depth
        self.trees = []; self.classes = None; self.class_weights = None
    def fit(self, X, y):
        self.classes = np.unique(y); n_samples = X.shape[0]; K = len(self.classes)
        counts = np.bincount(y.astype(int)); total = np.sum(counts)
        self.class_weights = {c: total / (K * counts[c]) for c in self.classes}
        self.trees = [[] for _ in range(K)]
        for k, cls in enumerate(self.classes):
            y_bin = np.where(y == cls, 1, 0); preds = np.zeros(n_samples); weight = self.class_weights[cls]
            for _ in range(self.n_estimators):
                prob = self._sigmoid(preds)
                gradient = weight * (y_bin - prob)
                tree = TreeEstimator(max_depth=self.max_depth)
                tree.fit(X, (gradient > 0).astype(int))
                preds += self.lr * tree.predict(X)
                self.trees[k].append(tree)
    def predict(self, X):
        scores = np.zeros((X.shape[0], len(self.classes)))
        for k in range(len(self.classes)):
            pred = np.zeros(X.shape[0])
            for tree in self.trees[k]: pred += self.lr * tree.predict(X)
            scores[:, k] = pred
        return self.classes[np.argmax(scores, axis=1)]
    def _sigmoid(self, x): return 1 / (1 + np.exp(-x))

class DecisionStump:
    def __init__(self):
        self.feature_idx = None
```

```
        self.threshold = None
        self.alpha = None
        self.polarity = 1

    def predict(self, X):
        n_samples = X.shape[0]
        X_column = X[:, self.feature_idx]
        predictions = np.ones(n_samples)
        if self.polarity == 1: predictions[X_column < self.threshold] = -1
        else: predictions[X_column > self.threshold] = -1
        return predictions

class AdaBoost:
    def __init__(self, n_clf=50): self.n_clf = n_clf; self.models = {}
    def fit(self, X, y):
        n_samples, n_features = X.shape; self.classes = np.unique(y)
        for c in self.classes:
            y_binary = np.where(y == c, 1, -1); weights = np.full(n_samples, 1 / n_samples); self.models[c] =
[]
            for _ in range(self.n_clf):
                clf = DecisionStump(); min_error = float("inf")
                for feature_i in range(n_features):
                    X_column = X[:, feature_i]; thresholds = np.unique(X_column)
                    if len(thresholds) > 10: thresholds = np.percentile(thresholds, [20, 40, 60, 80])
                    for threshold in thresholds:
                        p = 1; predictions = np.ones(n_samples); predictions[X_column < threshold] = -1
                        error = np.sum(weights[y_binary != predictions])
                        if error > 0.5: error = 1 - error; p = -1
                        if error < min_error: min_error = error; clf.polarity = p; clf.threshold = threshold;
clf.feature_idx = feature_i
                EPS = 1e-10; clf.alpha = 0.5 * np.log((1 - min_error + EPS) / (min_error + EPS))
                predictions = clf.predict(X)
                weights *= np.exp(-clf.alpha * y_binary * predictions); weights /= np.sum(weights)
                self.models[c].append(clf)
    def predict(self, X):
        scores = np.zeros((X.shape[0], len(self.classes)))
        for idx, c in enumerate(self.classes):
            clf_sum = np.zeros(X.shape[0])
            for clf in self.models[c]: clf_sum += clf.alpha * clf.predict(X)
            scores[:, idx] = clf_sum
        return self.classes[np.argmax(scores, axis=1)]

class GaussianNB:
    def fit(self, X, y):
        self.classes = np.unique(y); self.params = {}
        for c in self.classes:
            X_c = X[y == c]
            self.params[c] = {"mean": X_c.mean(axis=0), "var": X_c.var(axis=0) + 1e-9, "prior": len(X_c) /
len(X)}
    def predict(self, X):
        predictions = []
```

```
        for x in X:
            class_scores = []
            for c in self.classes:
                mean = self.params[c]["mean"]; var = self.params[c]["var"]; prior =
np.log(self.params[c]["prior"])
                likelihood = -0.5 * np.sum(np.log(2 * np.pi * var)) - 0.5 * np.sum(((x - mean) ** 2) / var)
                class_scores.append(prior + likelihood)
            predictions.append(self.classes[np.argmax(class_scores)])
        return np.array(predictions)
```

Appendix F - Training, Testing, and choosing the best

```
print(" REGRESSION EVALUATION")

X_train, X_test, y_train, y_test = train_test_split(X_all, y_val)
best_reg_models = {}
best_reg_score = -float('inf')
final_best_regressor = None
final_best_reg_name = ""


ols = LinearRegression()
ols.fit(X_train, y_train)
ols_preds = ols.predict(X_test)
ols_rmse = rmse(y_test, ols_preds)
print(f"OLS RMSE: {ols_rmse:.4f}")


lambdas_ridge = np.logspace(-2, 2, 20)
ridge_rmses = []
for l in lambdas_ridge:
    r = RidgeRegression(lambda_=l)
    r.fit(X_train, y_train)
    p = r.predict(X_test)
    ridge_rmses.append(rmse(y_test, p))

best_lambda_ridge = lambdas_ridge[np.argmin(ridge_rmses)]
print(f"Best Ridge Lambda: {best_lambda_ridge:.4f}, RMSE: {min(ridge_rmses):.4f}")

plt.figure(figsize=(6, 4))
plt.plot(lambdas_ridge, ridge_rmses, marker='o')
plt.xscale('log')
plt.title("Ridge: Lambda vs RMSE")
plt.xlabel("Lambda")
plt.ylabel("RMSE")
plt.show()

best_ridge = RidgeRegression(lambda_=best_lambda_ridge)
best_ridge.fit(X_train, y_train)
```

```
lambdas_lasso = np.array([0.01, 0.1, 1.0, 10.0])
lrs_lasso = np.array([0.0001, 0.001, 0.005, 0.01])
lasso_results = []

for l in lambdas_lasso:
    for lr in lrs_lasso:
        la = LassoRegression(lambda_=l, iterations=2000, learning_rate=lr)
        la.fit(X_train, y_train)
        p = la.predict(X_test)
        rmse_l = rmse(y_test, p)
        lasso_results.append((l, lr, rmse_l))

lasso_results = np.array(lasso_results)
best_idx = np.argmin(lasso_results[:, 2])
best_lambda_lasso = lasso_results[best_idx, 0]
best_lr_lasso = lasso_results[best_idx, 1]
print(f"Best Lasso Lambda: {best_lambda_lasso}, LR: {best_lr_lasso}, RMSE: {lasso_results[best_idx,
2]:.4f}")

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(lasso_results[:, 0], lasso_results[:, 1], lasso_results[:, 2], c=lasso_results[:, 2], cmap='viridis')
ax.set_xlabel('Lambda')
ax.set_ylabel('Learning Rate')
ax.set_zlabel('RMSE')
ax.set_title('Lasso: Lambda & LR vs RMSE')
plt.show()

best_lasso = LassoRegression(lambda_=best_lambda_lasso, iterations=5000, learning_rate=best_lr_lasso)
best_lasso.fit(X_train, y_train)


regressors = {
    "OLS": ols,
    "Ridge": best_ridge,
    "Lasso": best_lasso
}

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

for i, (name, model) in enumerate(regressors.items()):
    preds = model.predict(X_test)
    current_rmse = rmse(y_test, preds)
    r2 = r2_score(y_test, preds)

    if r2 > best_reg_score:
        best_reg_score = r2
        final_best_regressor = model
        final_best_reg_name = name
```

```
    min_val = min(min(y_test), min(preds))
    max_val = max(max(y_test), max(preds))
    axes[i].scatter(y_test, preds, alpha=0.5)
    axes[i].plot([min_val, max_val], [min_val, max_val], 'r--')
    axes[i].set_title(f"{name}\nR2: {r2:.4f}, RMSE: {current_rmse:.2f}")
    axes[i].set_xlabel("Actual")
    axes[i].set_ylabel("Predicted")

plt.tight_layout()
plt.show()

print(f"BEST REGRESSOR: {final_best_reg_name}")

print(" CLASSIFICATION EVALUATION")

X_train_c, X_test_c, y_train_c, y_test_c = train_test_split(X_all, y_exit)
classifiers = {
    "XGBoost": XGBoost(n_estimators=15, learning_rate=0.01, max_depth=5),
    "AdaBoost": AdaBoost(n_clf=15),
    "Gaussian NB": GaussianNB(),
}

best_clf_score = -1
final_best_classifier = None
final_best_clf_name = ""
n_classes = len(np.unique(y_test_c))

for name, model in classifiers.items():
    print(f"\nTraining {name}...")
    model.fit(X_train_c, y_train_c)
    preds = model.predict(X_test_c)

    acc = accuracy(y_test_c, preds)
    cm = confusion_matrix(y_test_c, preds, n_classes)
    precision = precision_per_class(cm)
    recall = recall_per_class(cm)
    f1 = f1_per_class(precision, recall)

    macro_f1 = macro_average(f1)
    kappa = kappa_score(y_test_c, preds, n_classes)
    min_class_acc = np.min(recall)

    print(f"Accuracy : {acc:.4f}")
    print(f"Macro F1       : {macro_f1:.4f}")
    print(f"Kappa Score    : {kappa:.4f}")
    print(f"Class Accuracies: {np.round(recall, 2)}")

    if min_class_acc < 0.05:
        adjusted_score = macro_f1 * 0.5
    else:
```

```
        adjusted_score = macro_f1 + (0.5 * min_class_acc)

    if adjusted_score > best_clf_score:
        best_clf_score = adjusted_score
        final_best_classifier = model
        final_best_clf_name = name

    plt.figure(figsize=(5, 4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=exit_map.keys(), yticklabels=exit_map.keys())
    plt.title(f"Confusion Matrix ({name})")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()

print(f"\nBEST CLASSIFIER: {final_best_clf_name}")
```

Appendix G - Demo

```
def predict_new_startup(startup_dict):
    df_new = pd.DataFrame([startup_dict])
    df_proc, _ = preprocess_pipeline(df_new, is_training=False, stats=saved_stats)
    X_new = df_proc.values.astype(float)

    val = final_best_regressor.predict(X_new)[0]
    exit_idx = final_best_classifier.predict(X_new)[0]
    exit_label = inv_exit_map[int(exit_idx)]

    return val, exit_label

new_startup = {
    "Startup Name": "QuantumHealth",
    "Industry": "IoT",
    "Funding Rounds": 4,
    "Funding Amount (M USD)": 249,
    "Revenue (M USD)": 25,
    "Employees": 600,
    "Year Founded": 1997,
    "Region": "Europe",
    "Market Share (%)": 4
}

print(f"DEMO FOR: {new_startup['Startup Name']}")

p_val, p_exit = predict_new_startup(new_startup)

print(f"Predicted Valuation      : ${p_val:,.2f} Million")
print(f"Predicted Exit Status    : {p_exit}")
```