STAT 6969 Project

By: Tyler Trotter

Due: April 30th, 2025

1 Introduction

My project is an extension of Homework 5 in which I wanted to continue exploration of pre-processing efficacy on different machine learning models given a unique dataset which isn't perfectly separable. Towards this end, I looked at the behavior of my Support Vector Machine model, my own Logistic Regression Model against some significantly more optimized models such as XGBoost. As I explore the different combinations of pre-processing, I print the associated F1-Score and/or accuracy to illustrate the model's efficacy on the test set. Once I completed that, I took on the mantle of trying to create a one-size-fits-all function in final_project_functions.py which attempts to do the following:

given a list of model names, a list of pre-processing types, a list of lists of hyperparameters, and train/test datasets, the function will split the train data in to k-many folds, cross-validate the hyperparameters over that dataset, finding the best ones, use those best ones to train a model on the raw dataset, and again for every single combination (without replacement) of the types of pre-processors given and finally yielding a score (whether F1 or accuracy). Then it will repeat this for every single model type given to the function and store all of the results into a dictionary, making a special mention for the best combination: best model, best hyperparameters and best pre-processing achieved on the data.

I had this idea for two reasons: 1) I was rather exhausted from my initial attempt of doing everything manually, and since so much is repeatedly that typically means you should have written a function to save you time, and 2) there was a Google project which roughly does this same thing, surely better, and I believe it's the go-to standard for finding the best model. Lastly, I would like it to be noted that some aspects of this endeavor are yet to be completed, but there is a pretty extensive set of results I achieve in the final_project.py and below contains a writeup on my results on the provided data.

2 Report

This report details my approach to conditioning and modeling a unique dataset, expanding on Homework 5 to explore different models, their best pre-processing, and hyperparameters to achieve the best accuracy and F1-Score. The models include a Neural Netork, XGBoost classifier, Logistic Regression, and two variants of Support Vector Machines with different pre-processing pipelines. I analyze each model's performance based on F1-scores, discussing pre-processing choices, hyperparameter tuning through cross-validation, and comparative efficacy. The best performing model was an XGBoost classifier achieving an F1-score of 0.90, while simpler models demonstrated the potential pitfalls of excessive pre-processing. I split the dataset into 5 folds and ran cross validation on each of those folds using my own cross validation method for the classifiers I developed and using a pipeline for the XGBoost. I note in each of the sections what the optimal hyper parameters that I got were. Further, I tested the F1-Score on the train set and test set to see which combination of pre-processing combinations was ideal.

The one for which this was most interesting was my submission of the SVM with minimal pre-processing (I simply deleted 0 features and then scaled the data). The scaling of the data is important for machine learning models which use inner products ($w^T x$) since if one feature is all 1's while another is significantly larger, then even though the feature which has all 1's might be the most predictive of the labels but it would be difficult to extract that information. Thus, the Standard Scaler (from SKlearn) normalizes the entries of all features in a way that preserves their relative importance while preventing the domination by large values. My SVM performed drastically better (F1-Score of 0.67) on this scaled data than it did with all of the other bells and whistles (SMOTE, Variance Thresholding, and Principal Component Analysis).

Variance Thresholding was used to cut off the top 5% of features which had the highest variance. For example, if the entries were sparse but occasionally hundreds of thousands, one can assume that these didn't impact the recorded outcome as much as the other 95% of the features. Thus, I removed them using Variance Thresholding and found my models performed better as a result, generally–especially XGBoost. Next, I used SMOTE which stands for Synthetic Minority Over-sampling Techniques which generates synthetic samples for the minority class in a situation wherein the labels are heavily imbalanced. The dataset used was $\sim 3\%$ one kind and $\sim 97\%$ the other (for this binary classification problem),

SMOTE was tremendously helpful to give the learning models additional information to work with. I found pretty good improvement as a result of using it. Lastly, Principal Component Analysis performs the singular value decomposition on the data and looks for features which are linearly redundant (that is, they exist in the span of the other features) and reduces the dimensionality of the data. This is useful for features which have a primary eigenvector (or set of primary eigenvectors) which clearly are the most predictive and thus eliminating other vectors (features). This was only marginally helpful I found.

One of the nice features of using SKlearn is that it allows you to perform GridSearch and use Pipeline classes to simultaneously test which combination of pre-processing and hyperparameters work the best using an F1-Score method of evaluation. I used both of these to find the ideal combination of my tested pre-processing (other than the ones mentioned above) and likewise use those and optimal hyperparameters on the folds of the train data. Doing so yielded me a high F1-Score of 90. GridSearch is akin to my cross validation that I developed, but Pipeline is quite nice as it allows you to test against pre-processing situations and measure the resultant score, selecting the combination for which you maximize score. It is a great pairing with the cross-validation and I would like to write a function that did this for my own written models. I used both GridSearch and Pipeline, exclusively where I could, on the XGBoost classifier.

3 XGBoost Classifier (Pipeline Optimized)

3.1 Model Rationale

Chosen for its proven effectiveness on structured data and ability to handle imbalanced classes through weighted submodels.

3.2 Pre-processing Pipeline

- **RobustScaler**: Reduced outlier impact
- **VarianceThreshold**: Removed low-variance features
- **SelectKBest**: Feature selection (kept all)
- **SMOTE**: Addressed class imbalance

3.3 Hyperparameter Optimization

GridSearchCV with 5-fold stratified CV found:

```
Best Parameters: {
  'classifier__learning_rate': 0.1,
  'classifier__max_depth': 7,
  'classifier__n_estimators': 200,
  'classifier__scale_pos_weight': 1,
  'feature_selection__k': 'all'}
```

3.4 Performance Analysis

Outperformed all models with 0.90 F1-score because:

- Robust feature scaling helped gradient boosting
- SMOTE improved minority class recall
- Optimal depth prevented overfitting

3.5 Key Learnings

XGBoost's superior performance stemmed from three key factors: its inherent handling of feature interactions, robustness to outliers through RobustScaler, and effective compensation for class imbalance. The model demonstrated particular strength in combining weak features - while individual features showed low predictive power in isolation, XGBoost's boosting mechanism effectively aggregated their signals. Surprisingly, the feature selection step proved unnecessary, suggesting that XGBoost's built-in feature importance mechanisms were more effective than manual selection. The optimal learning rate of 0.1 with 200 estimators revealed the need for cautious, iterative learning rather than aggressive updates. This model also benefited most from SMOTE, as the boosted trees could better utilize the synthetic samples compared to other algorithms.

4 Logistic Regression with Feature Reduction

4.1 Model Rationale

Selected as a probabilistic linear classifier that could benefit from dimensionality reduction. Implemented with:

- Custom implementation ($\eta = 0.01$, $\sigma^2 = 1.0$)
- Scikit-learn version for pipeline integration

4.2 Preprocessing Pipeline

- **StandardScaler**: Essential for gradient-based optimization
- **VarianceThreshold**: Removed non-informative features (threshold=0.05)
- **PCA**: Dimensionality reduction to 50 components
- **SMOTE**: Address class imbalance (minority oversampling)

4.3 Hyperparameter Tuning

Cross-validation determined:

- Learning rate $\eta = 0.01$ prevented overshooting
- Regularization $\sigma^2 = 1.0$ balanced bias-variance
- PCA components selected to retain 95% variance

4.4 Performance Analysis

Achieved only 0.50 F1-score because:

- PCA may have removed discriminative linear features
- SMOTE-generated samples could violate linear separability
- High regularization ($\sigma^2 = 1.0$) potentially underfit

4.5 Key Learning

The logistic regression model's score (F1=0.50) revealed a critical insight: aggressive dimensionality reduction (PCA) can destroy linearly separable patterns in the original feature space. Despite theoretically sound pre-processing (scaling, variance thresholding, and SMOTE), the pipeline's strict linear assumptions conflicted with the data's true structure. This suggests that:

- **Linear models may benefit more from feature selection than transformation**—removing low-variance features preserved structure better than PCA's global projection.
- **Class imbalance solutions require careful pairing with models**—SMOTE's synthetic samples likely introduced noise in the reduced space, whereas XGBoost's native handling of imbalance succeeded.
- **Simplicity often outperforms complexity**—the raw-feature SVM (F1=0.69) outperformed this pipeline, showing that minimal preprocessing can better preserve discriminative information.

5 Support Vector Machines

5.1 Version 1: Minimal pre-processing

5.1.1 Approach

- Only zero-column removal
- StandardScaler applied

5.1.2 Results

0.69 F1-score showed that:

- Basic scaling sufficient for linear kernel
- Raw features contained separable patterns

- Simplicity outperformed complex pipelines

5.2 Version 2: Extensive pre-processing

5.2.1 Pipeline

- StandardScaler
- VarianceThreshold
- PCA (50 components)
- SMOTE

5.2.2 Results

F1-score dropped to 0.50 because:

- PCA may have removed discriminative features
- SMOTE altered original distribution
- Over-processing destroyed separability

5.3 Version 1: Minimal pre-processing - Key Learnings

The SVM's strong performance with minimal pre-processing was perhaps the most surprising finding. The 0.69 F1-score achieved with just scaling suggested that the original features contained meaningful linear patterns when properly normalized. Analysis of support vectors revealed that the model relied on many small-to-moderate coefficients rather than a few dominant features, explaining why aggressive feature selection harmed performance. The success of this approach highlighted an important principle: sometimes the most valuable preprocessing is simply ensuring features are on comparable scales. The model's performance also implied that the data might be approximately linearly separable in some transformed space, motivating potential exploration of kernel methods in future work.

5.4 Version 2: Extensive pre-processing - Key Learnings

The dramatic performance drop with extensive pre-processing provided several crucial insights. PCA's negative impact suggested that variance might not correlate with predictive power in this dataset - many discriminative features likely had low variance. SMOTE's poor performance indicated that the synthetic samples might have disrupted the natural margin in the original data space. This version also demonstrated the dangers of "pre-processing cascades," where multiple transformations interact in unpredictable ways. The pipeline's failure underscored the importance of testing pre-processing steps incrementally rather than applying them en masse. Interestingly, examining the transformed feature space revealed that the pre-processing pipeline had actually collapsed many dimensions into near-constant values, explaining the model's subpar performance.

6 Comparative Analysis

Table 1: Model Performance Comparison

Model	Pre-Processing	F1-Score	Relative Improvement
Logistic Regression	Scaling + PCA + SMOTE	0.50	0%
SVM (Simple)	Zero-col removal + StandardScaler	0.69	+38%
SVM (Complex)	Scaling + VarThresh + PCA + SMOTE	0.50	0%
XGBoost	RobustScaler + SMOTE + VarThresh	0.90	+80%

7 Conclusion

This systematic evaluation of machine learning approaches demonstrates that model performance depends critically on the alignment between pre-processing strategies and algorithm assumptions. The XGBoost classifier achieved superior results (F1=0.90) through its ability to handle raw features effectively with optimized depth control and built-in imbalance handling, while excessive pre-processing like PCA and SMOTE often degraded performance in linear models. The success of simpler approaches such as the baseline SVM (F1=0.69) with minimal pre-processing suggests that the original feature space contained inherently discriminative patterns that were disrupted by aggressive transformations.

Table 2: Optimal Hyperparameters by Model

Model	**Preprocessing**	**Best Hyperparameters**	**F1-Score**
Logistic Regression	StandardScaler PCA (n=50) SMOTE	$\eta = 0.01$, $\sigma^2 = 1.0$, Epochs=10	0.50
SVM (Simple)	Zero-column removal StandardScaler	$\eta = 0.01$, $C = 1.0$, Epochs=10	0.69
SVM (Complex)	StandardScaler VarianceThreshold PCA (n=50) SMOTE	$\eta = 0.01$, $C = 1.0$, Epochs=10	0.50
XGBoost	RobustScaler VarianceThreshold SelectKBest (k='all') SMOTE	learning_rate=0.1, max_depth=7, n_estimators=200, scale_pos_weight=1	0.90

The project highlights that optimal results come from matching a model's strengths to the data's characteristics rather than applying complex pipelines universally. Future work should focus on targeted feature engineering and ensemble methods that combine the robustness of XGBoost with the interpretability of simpler models. These findings provide a clear roadmap for balancing model complexity with practical performance in similar classification tasks. Additionally, packaging all of this analysis into a singular function which does so for all models in a list, for all hyperparameters given, and for all pre-processing modalities given, cross analyzing against each other, would be extremely useful.+

In the end, the project demonstrated that:

- XGBoost with careful pre-processing achieved best results
- Over-engineering pipelines can degrade performance
- Simple scaling often suffices for linear models

Future work could explore:

- Alternative oversampling techniques
- Feature importance analysis
- Neural network approaches