ST443 Project

46930 42502 48397 50691

Abstract

We analyze two distinct datasets. In the first, we achieve very high F1 scores (0.95) using a variety of models but we are unable to consistenly improve to higher scores so we use SVM as our final predictor as it perfoms well throughout. In the second, we use a variety of different approaches and achieve very positive results, (0.9, 0.94) balanced accuracy and (20,291) features with Lasso-SVM and Anova-SVM, respectively.

T1.1 Exploratory data analysis

Basic facts

We have a dataset that consists of 5471 (n) samples with 4124 (p) columns. The features of our data are all continuous, in a logarithmic scale and they consist of expression levels for genes. It is worth mentioning that the dataset is sparse, that is, a lot of cells have many gene expressions that are 0 [1].

We have 2 distinct classes of cells, the TREG cells and the CD4+T cells. Our classification task consists of prediciting the cell type. We have some class imbalance, the ratio is 6/10 in favour of the CD4+T, which is the dominant class. We try the option of tuning the threshold for classifying to one cell or another. By default the probabilistic models in Scikit-learn classify to the positive class if the conditional probability for the given model $\mathbb{P}(y|X) > 0.5$[2]. We will tinker with this threshold to try to optimize the F1-Score given our class imbalance.

Visualization and dimensionality redcution

Both the fact that $n \approx p$ and the fact that the data is sparse, point us towards using regularization and feature selection. The instructions for this problem also make us use PCA with 10 components. To confirm whether the number of components is optimal, we can plot the cumulative sum of the explained variance by each of the components. While the plot is relegated to the notebook, choosing the first ten components makes us use only 9 percent of the original variation. In Section we try to tune the number of components to get an improved F1-Score.

For completeness, we add Figure 1 where we use t-SNE to reduce the dimensionality of the data from 4124 to 2[3]. We do this with the purpose of visualizing the joint distribution (after the t-SNE transformation) of both cells. Figure 1 shows some separability between the two classes. It also shows some cells of a given class (TREG) in regions where the density is much higher for the other type of cell (CD4T), maybe we can use this as intuition as to why in later sections we find it difficult to improve the F1-Score beyond 0.95.

[1]While it is hard to have an objective measure of sparsity we can compute the percentage of zeros for each type. It is high for most genes. The plot is relegated to the notebook.

[2]See scikit references on threshold tuning.

[3]The underlying algorithm is stochastic and quite sensitive to how we tune the hyperparameter of perplexity

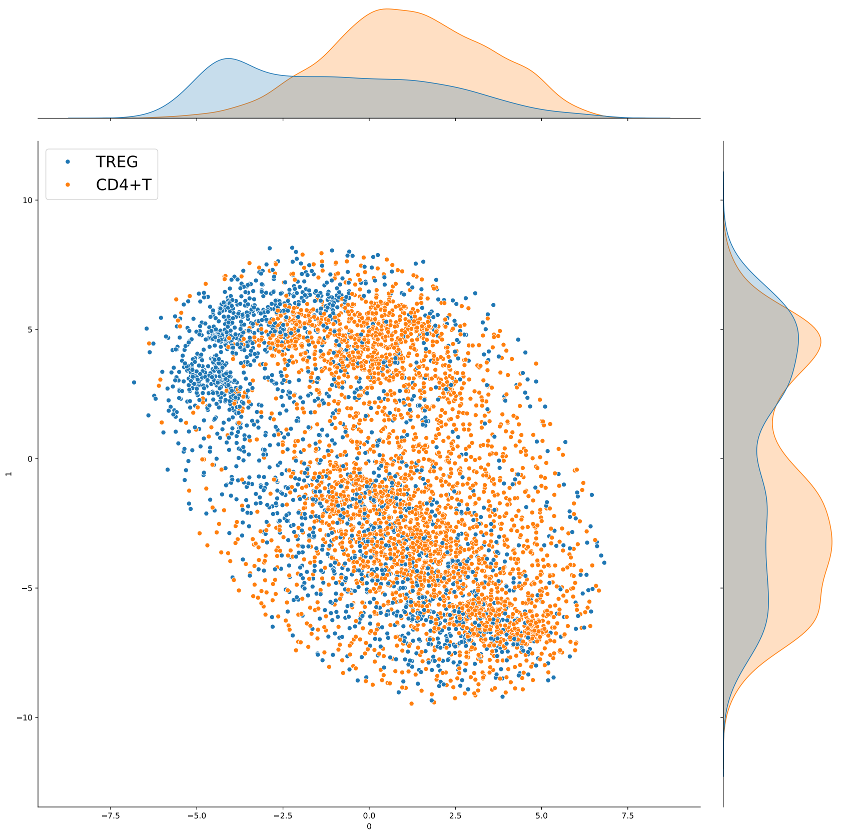

Figure 1: Joint distribution of transformed data by t-SNE

T1.2 Training, tuning and evaluating baseline models

This section is split into two parts, the first part describes our code from Python at a high-level. It focuses on some of scikit-learn procedures to streamline the analysis and guarantee good test performance.[4] Then we move to explain which parameters we tune to improve the performance of the baseline models.

Using the scikit-learn toolkit

Because we implement hyperparameter tuning we first need to split the data into a test and train subset -this is needed because we tuned some hyperparameters, if not an estimate of test error obtained through cross validation should be valid -[5]

To do the hyperparameter tuning we implement grid search[6] which is a brute force approach that consists of trying all hyperparameter combinations specified by the researcher. We use the training set for a given set of parameters, then use cross validation -a part of our training data is left out as a validations set, the model is trained on the rest of the training data and we get a validation error estimate, this process is repeated 5 times - and obtain an estimate of the metric we are trying to optimize over. We iterate over all parameter combinations and pick the one that maximizes the metric we choose (in Task 1 it was F1 score).

[4]We believe learning how to use scikit learn ”properly” was one of the most valuable things we learnt in Task 1.

[5]See scikit references on train, test and hyperparameter tuning.

[6]See scikit referemces on grid search.

Some models need the data to be standarized, and we are also asked to use PCA, we therefore make use of pipelines [7], which are a sequence of data pre-processing steps with a final estimator. Using pipelines streamlines our analysis and avoids some common pitfalls. Example of a pipeline: Our cross validation estimator has selected a random subset of the training data, then the pipeline will: Standarize the data $\rightarrow$ Apply PCA $\rightarrow$ Fit the classifier on the preprocessed data.

Finally, throughout our analysis we tried to set a random seed so that our results are reproducible. However we forgot about randomness in the PCA algorithm used by scikit-learn, while the exact numbers may change, the overall picture and decision making is not affected.

Hyperparameter tuning the baseline models

We focus on tuning the models without PCA, as that is where most of the gains can be made. Due to the relatively high dimensionality of the data, some models are underperfoming if we use all the features avalable. In contrast even the out-of-the-box models without hyperparameter tuning perform really well when training on the first 10 components of the PCA. Therefore we compare the tuned models without PCA to the untuned models with PCA. Next we show what we tuned and the results we obtained.

What did we tune?

For the first 3 models *-LDA, Logistic regression* and *QDA*- the tuning is mostly focused on imposing strong penalization mechanisms, such as lasso, ridge or elastic net.

For *k-NN*, there are a lot of interesting parameters. We search across different numbers of neighbors. We also tune across rules to determine who the nearest neighbors are. To achieve this we use the Minkowski(p) distance, this is a distance that nests the classical euclidean (Minkowski(p_1)) and manhattan distances (Minkowski(p_2)), so we search across p to try different distances. Finally we also try to compute the decision rule using different weights, the default where all neighbors have the same weight in the vote and one where the votes from the closest neighbors within the neighborhood carry more weight.

For *Support Vector Machine*, we tune across different kernels and regularization strenghts. The default uses the radial/gaussian kernel, we also try to use linear kernels as well as some others we haven't seen in class.

For *Random Forests* our grid search is not extensive, we tune the criterion made to determine the quality of a split (gini and entropy), the number of trees in the forest and the number of features a tree in the forest will choose from. It is worth mentioning that we consider pruning, as we saw the method in class, however we decide against it as it would be computationally expensive.

For *Gradient Boosted Decision Trees (GBDT)* we consider different loss functions, number of iterations, learning rates, as well as the depth and features used by the trees. We also set an early stopping rule as this grid search is computationally expensive. For other models the tuning parameters are mostly self explanatory, in this case it is more complex. For intuition let us remember that this is similar to AdaBoost, learning determines how new iterations influence the final vote and the tree complexity should be tuned to avoid overfitting. If we run too many iterations with a deep tree we might overfit, while if we run too few iterations on shallow trees, our performance may be suboptimal. Finding the optimal between iterations, learning and tree complexity is hard.

Finally, across our tuning, we use tuning paramters related to class weights. As while the classes are not as imbalanced as in Task 2, there is some imbalance in favour of the CD4+T cell type.

Analysis of results

We compare our fully tuned models without PCA to out of the box models with PCA. As you can see from Table 1, out-of-box with PCA have similar or better performance to the ones we fully tune without PCA. The models most affected by PCA are LDA, QDA and *k-NN*. Using PCA improves the perfomance of QDA and K-NN dramatically. For Logistic regression and LDA, the performance is not improved too much. That is because the optimal models from the hyperparameter tuning already contain severe forms of regularization, which is very similar in spirit to what PCA is doing. SVM

[7] See scikit references on pipelines.

is consistently great. Random forest gets a small improvement from the PCA and GBDT lowers the performance slightly, although both these models are consistent in making very good predictions.

Model	Accuracy	Balanced Acc.	AUC	F1 Score	Confusion Matrix
LDA	**0.965**	**0.956**	0.993	**0.951**	$[687, 6], [32, 370]$
LOGIT	0.958	0.954	0.993	0.942	$[672, 21], [25, 377]$
QDA	0.367	0.500	0.500	0.537	$[0, 693], [0, 402]$
KNN	0.788	0.777	0.874	0.718	$[567, 126], [106, 296]$
SVM	0.956	0.953	0.993	0.940	$[669, 24], [24, 378]$
RF	0.943	0.924	0.993	0.917	$[690, 3], [59, 343]$
GBDT	0.963	0.955	**0.994**	0.949	$[684, 9], [31, 371]$
With PCA					
LDA	0.950	0.936	0.994	0.928	$[685, 8], [47, 355]$
LOGIT	0.966	0.965	0.994	0.954	$[672, 21], [16, 386]$
QDA	0.960	0.958	0.993	0.946	$[669, 24], [20, 382]$
KNN	0.928	0.911	0.956	0.896	$[675, 18], [61, 341]$
SVM	**0.967**	**0.966**	**0.994**	**0.956**	$[672, 21], [15, 387]$
RF	0.957	0.947	0.989	0.940	$[683, 10], [37, 365]$
GBDT	0.947	0.938	0.990	0.926	$[673, 20], [38, 364]$

Table 1: Fully tuned models with PCA vs out-of-box with PCA.

T1.3 Our 3 models

We try 3[8] simple approaches guided by models we saw in class and from our experience with the baseline models. We implement AdaBoost, tinker with the decision thresholds and try different numbers of PCA components with an out-of-box SVM. None of our approaches let us **consistenly** improve to F1 scores of 0.96 or 0.97 [9].

Tuned AdaBoost

We tune an AdaBoost model with different baseline estimators and changes in the parameters related to learning rate and others. The most interesting modification is the choice of baseline estimator, we used the default stubs, trees with a maximum depth of 2 and logistic regression. Across these methods the logistic regression is the baseline estimator with best results, having said this, it also required a small number of iterations and a low learning rate. This can be due to the fact that as Logistic regression is a more powerful base estimator, it can be prone to overfit.

Threshold tuning

Because even out-of-box models where giving very good results we tried to think out of the box and tune something completely different. All models in scikit-learn classify to a given label when a conditional probability is larger than 0.5 (something similar happens for non probabilistic models). Given our class imbalance we can use a trained model and use cross validation to optimize over this threshold. In Figure 2 we can see how it improves the F1 score of a random forest. While this approach yields great results in Task 2, here our estimators were very robust to threshold modification, meaning the F1 scores are mostly flat around the default threshold.[10]

[8]We tried more than 3 but settled on these 3 because they showcase very different methods.

[9]Due to this we omit showing these results, they can be found in the notebook for completeness.

[10]This could have to do with how we grid searched -trying to maximize F1 score- or due to the imbalance between classes not being too large. The code for the graph below is taken and modified from the following towardsdatascience article.

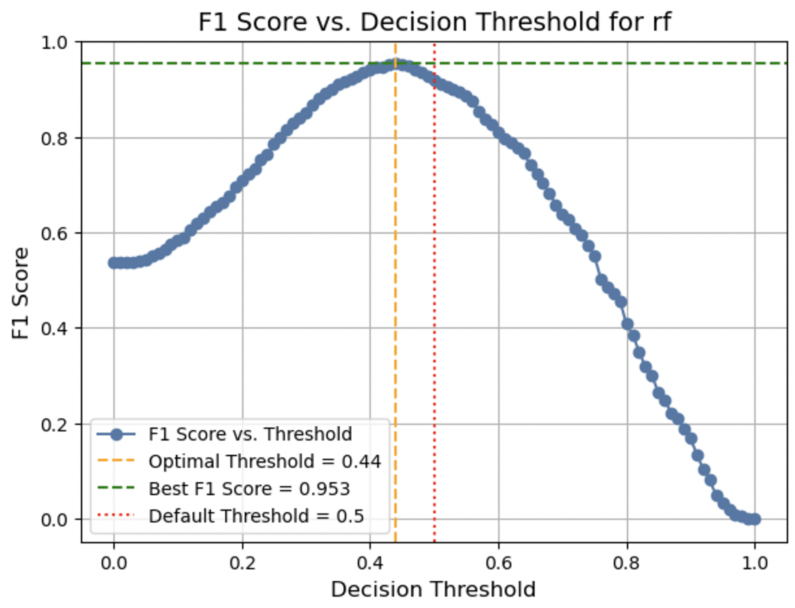

Figure 2: Thresholding example with a random forest model. F1 scores come from cross validation.

Tuning the number of principal components

For this minimalistic approach we take an out-of-box SVM (radial kernel) and we tune the number of principal components it takes as an input. We wanted to focus on this one dimension of the analysis on one of our best performing models. The optimal number of components was 22 but it didn't yield us greatly improved results. Another option would have been to use a model that struggled without PCA and to see how its performance changed.

T1.4 Final predictor

Many models give very similar F1 scores, these models are very different in nature. And due to the inherent randomness we can't be sure which model will work best in the the left out test sample. The biggest thing we can control is to train the model on **all** the data we have available. For this final predictor the full dataset we have available becomes our training set. We use an out-of-box SVM with 22 principal components as our final estimator, this is because the SVM has given us consistenly great results throughout our analysis.

T2: Data

The dataset is a two-class classification data set with binary features describing the three-dimensional properties of a molecule in a compound. The data originally had 50,000 features, but 50,000 additional random probes have been added. The final data is high-dimensional in the features scale as it has 800 observations (n) and 100,000 features (p).

Goal

This part of the project aims to train feature selection models on the 100,000 features and choose the best model that increases the balanced accuracy while retaining the number of selected features at a minimal level.

T2.1 Exploratory Data Analysis

Some initial exploratory analysis has been performed on the data to help understand the structure, distributions and key characteristics about the data.

- Balance: The data is very imbalanced: as shown in Figure 3, 90.25% of the observations belong to class -1. Based on this fact about the data, the metric used to assess all the models is balanced accuracy. Nevertheless, for only one of the models, we used the Synthetic Minority Oversampling Technique, or SMOTE, to treat the imbalance before training the model, as we found that this gives a better balanced accuracy for this specific model.

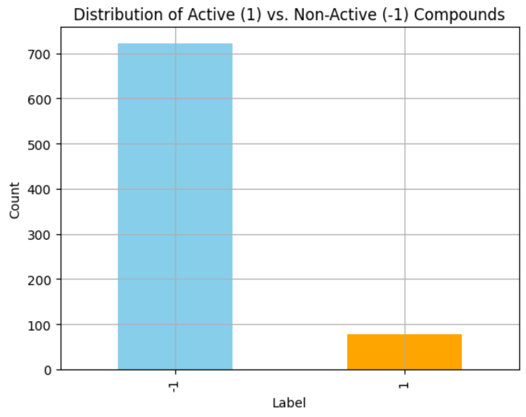

Figure 3: Class imbalance.

- Sparsity: The data is very sparse; 99.09% sparsity means that 99.09% of the dataset consists of zeros.
- Variability: Despite the data having no missing values, the summary statistics of the data showcased that some of the columns have zero variance (meaning the column has the same value across all the observations), other columns have relatively very small variances (only around 30% of the features have variance > 0.01. Thus it's worth noting that these features are removed for only some of the models that don't inherently rank features based on the variance, so as to improve the computational efficiency of the specific models, we removed these features before the training.

Data Preprocessing

The data is split into training (80%) and test data (20%). The models are trained on the training set, and all the balanced accuracies are obtained from the test set's predictions.

T2.2

Random Forest with a forward stepwise selection

For this classification task, we applied a Random Forest classifier to the reduced set of 30,000 features, ranking them by importance. Using forward stepwise selection on the top-ranked 1,000 features, we identified an optimal subset of **43 features**, achieving a balanced accuracy of **0.84**.

The model exhibited a low false positive rate (FPR), with only 9 negative samples misclassified as positive, and a low false negative rate (FNR), with just 4 positive samples misclassified as negative.

Random Forest handles class imbalances by assigning balanced class weights, adjusting the miss-classification penalty for each class during training. It is also resilient to overfitting by employing bagging and averaging predictions across multiple decision trees, reducing variance.

Random Forest was effective for our project due to its ability to handle class imbalances, manage high-dimensional data, reduce overfitting, and maintain high performance.

Lasso Regression

Lasso regression with threshold tuning is used in this model for feature selection. Lasso selects the most relevant features by shrinking less important feature's coefficients to zero. It also reduces the risk of overfitting on the correlated features.

Lasso predictions are by default transformed into class labels with a default threshold equal to zero. With this threshold, we hypertuned alpha on the training set using 5-fold cross validation; the best-balanced accuracy we could obtain was 0.8.

We anticipated that changing the threshold would improve the overall balanced accuracy, so after providing the lasso probabilities that each observation belongs to class 1 or -1, a 5-fold cross-validation was used to perform hyperparameter tuning on both alpha and the threshold. The optimal threshold for each alpha is calculated by averaging the thresholds over the folds.

A final LASSO model is then trained on the whole training set with the best alpha (defined as the one that yields the best cross validated balanced accuracy) and the overall balanced accuracy is obtained from the test set. A balanced accuracy of **0.8507** was achieved and only **181 features** were selected.

Support Vector Classifier

For this model, we developed a hybrid method with both Lasso and SVM. Initially, Lasso is used to rank the features and subset the training data (by eliminating features with a lasso coefficient equal to zero). Then, a Support Vector Classifier with an RBF Kernel is trained for the classification. The choice of this kernel is for its ability to handle complex decision boundaries for non-linearly separable data. Also, SVC automatically handles imbalanced data by adjusting weights inversely proportional to class frequencies in the input data when the class_weight parameter is set to balanced.

A 5-fold hyperparameter tuning on the training set was performed to obtain the best alpha. After repeating this process for a defined alpha space, the best alpha is used to train a final SVC model on the whole training set and the test set is predicted, with this approach we obtained a balanced accuracy of **0.8924** with a very minimal number of features **20 features** selected. This hybrid method capitalized on the strength of both Lasso and SVM, efficiently reducing the high dimensionality of the data and addressing the feature space complexity and the imbalance, resulting in highly balanced accuracy.

T2.3

Logistic Classification with Elastic Net Penalty

We selected a logistic classification with Elastic Net penalty for this task because it combines the properties of both Ridge and LASSO regression, offering unique advantages. Unlike LASSO, which tends to select only the most relevant features and exclude correlated ones, Elastic Net allows for the inclusion of a larger number of non-zero features when the number of features p exceeds the number of observations n. This is particularly beneficial in high-dimensional datasets like ours.

Elastic Net balances by selecting groups of highly correlated variables while still enforcing sparsity, unlike standard Ridge regression or pure LASSO. This ability to handle multicollinearity effectively—while maintaining a simpler and more interpretable model—is a key advantage over using LASSO alone.

We need to maximize this log-likelihood function in order to find the coefficients for the logistic classification :

$$\sum_{i=1}^{N} \log\left(\frac{\exp(\beta_{0g_i} + x_i^T \beta_{g_i})}{\sum_{j=1}^{K} \exp(\beta_{0j} + x_i^T \beta_j)}\right) - \lambda \sum_{k=1}^{K} \sum_{j=1}^{p} \big(\alpha|\beta_{kj}| + (1-\alpha)\beta_{kj}^2\big).$$

The second term is the penalty with α the proportion of Ridge and Lasso that we want and λ is the penalty parameter.

We tuned the two hyperparameters for this model: $\alpha = 0.3$ and $\lambda = 100$. The parameters show that we are closer to the Ridge regularization with a strong penalty regularization.

The Logistic Regression model with an Elastic Net penalty achieved a balanced accuracy of **0.83**, using only **29 selected features**. The confusion matrix confirmed strong predictive performance, with only a few misclassifications. This demonstrates that the feature selection process was both effective and efficient, striking the right balance between simplicity and accuracy.

Nearest Shrunken Centroids classification

The Nearest Shrunken Centroids (NSC) method is a classification approach designed to work effectively with high-dimensional data, where the number of features is much larger than the number of observations. When p larger than n, fitting a standard linear discriminant analysis (LDA) becomes impractical because LDA uses all features (genes) for classification.

The regularization terms in NSC assume that the features are independent within each class, that is, the within-class covariance matrix is diagonal. Even though that the features will rarely be independent within a class, when p is larger than N we don't have data to prove their dependencies. The assumption of independence greatly reduces the number of parameters in the model and often results in an effective and interpretable classifier.

The NSC method calculates the discriminant score d_{kj} for each feature (j) in class (k), these scores are then shrunk towards zero using a threshold δ. The shrunken centroids $\bar{x}'_{kj}$ are then plugged in the discriminant function of the LDA classifier $\delta_k(x^*)$. The new observation is assigned to the class with the highest discriminant score.

$$d_{kj} = \frac{\bar{x}_{kj} - \bar{x}_j}{m_k(s_j + s_0)}, \quad d'_{kj} = \text{sign}(d_{kj})(d_{kj} - \delta)_+, \quad \bar{x}'_{kj} = \bar{x}_j + m_k(s_k + s_0)d'_{kj},$$

$$\delta_k(x^*) = -\sum_{j=1}^{p} \frac{(x^*_j - \bar{x}_{kj})^2}{s_j^2} + 2\log\pi_k$$

We tuned the delta parameter during this process to optimize classification performance. After applying NSC we still had 3000 features, so we performed a feature selection using a quartile threshold based on balanced accuracy through cross-validation. This process effectively reduced the feature set to **225 feature**s, achieving a balanced accuracy of **0.86**. This final feature set demonstrates both improved interpretability and strong classification performance.

ANOVA F-test

ANOVA, or the analysis of variance, is a univariate statistical test that selects features with strong class-discriminative power. It uses the f-test to analyse differences between means, measuring the between-group variability to within-group variability to assess whether values of each feature differ significantly between target classes.

The method is implemented using sklearn's **SelectKbest**, a filter-based method, with the scoring function set to **f_classif** to evaluate the feature importance based on ANOVA.

Initially, we removed all the features with zero variance to ensure the bypass of division by zero when calculating the f-score. Then, we iterate in the feature space to hyper-tune the optimal number of features to be selected. An SVC is trained on each subset of the selected features, and the cross-validated balanced accuracy is recorded.

The optimal number of features (yielding the highest cross-validated accuracy) is then used to train a final SVC model, and the test predictions are obtained. This method produced the highest balanced accuracy of **0.9375** with **291 features**. Despite having a very simple concept, this method focuses on the strong correlation between the features and the target variables, selecting only informative features for the target variable.

Results

We used a blend of filter, wrapper and embedded methods for our models. All models performed well and selected less than 300 features (around 0.3% of the original feature space). The balanced accuracies of the models varied, with ANOVA obtaining a balanced accuracy of 0.9375. In terms of the minimal number of selected features the support vector classifier model only selected 20 features. Moreover, and because this problem is high dimensional on the number of features, we also recorded the computation time as some models are more computationally expensive than others. Logistic Elastic Net, Lasso and SVC took more time compared to the other models, with Nearest Shrunken Centroids taking the least time of less than 2 mins. We selected the support vector classifier as our best model because it efficiently classifies the test data using only 20 features and achieves a balanced accuracy of 0.89, all while requiring relatively little computational time.

Model	**Balanced Accuracy**	**Number of features**	**AUC**
Random Forest with stepwise selection	0.84	43	0.85
Lasso Regression	0.85	181	0.93
Support Vector Classifier	0.89	**20**	0.92
Logistic Classification with Elastic net	0.83	29	0.84
Nearest Shrunken Centroids	0.87	221	0.87
ANOVA F-value	**0.94**	291	**0.97**

Table 2: Performance Summary for All the Models

1 References

Hastie, T., Tibshirani, R., Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (2nd ed.). Springer. `https://doi.org/10.1007/978-0-387-84858-7`

Hastie, T., Tibshirani, R. (n.d.). *An Introduction to Statistical Learning with Applications in R. Retrieved from https://hastie.su.domains/ISLP/ISLP*$_{w}ebsite.pdf.download.$

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830. Retrieved from https://scikit-learn.org/stable/