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.

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).

[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

[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.

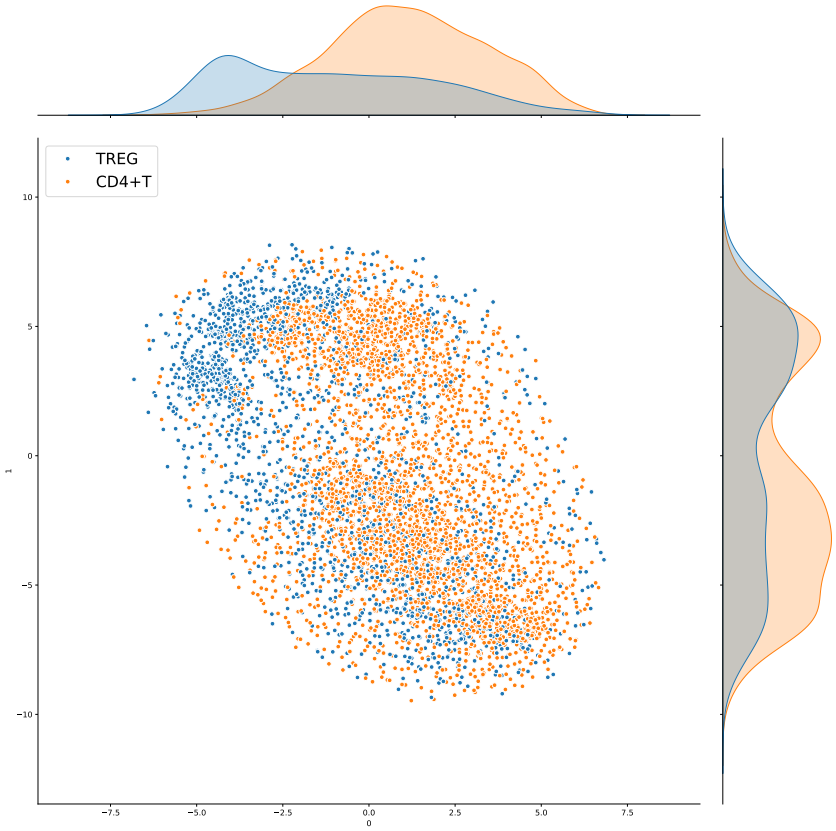

Figure 1: Joint distribution of transformed data by t-SNE

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

[7] See scikit references on pipelines.

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 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]

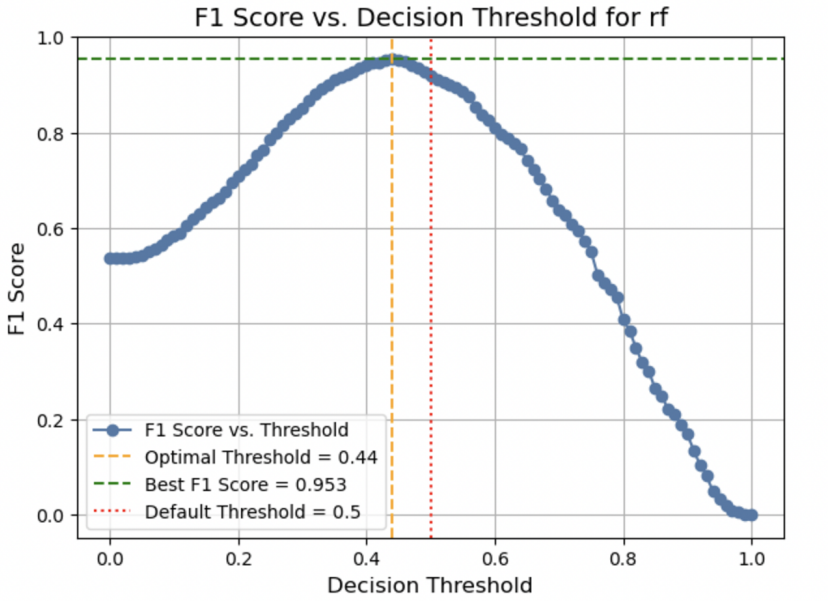

Figure 2: Thresholding example with a random forest model.

[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.

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.