Machine Learning mini-project

Fall semester 2023

BIO-222 / Brea Johanni Michael

Élise Boyer
elise.boyer@epfl.ch
@elboyer228

Johann Clausen
johann.clausen@epfl.ch
@johanncc01

1. Introduction

This project focuses on predicting drug retention times in liquid chromatography using machine learning. Retention time (RT), unique to each drug, depends on its chemical properties and the chromatography setup. Data on drug structures and retention times from different labs and platforms has been gathered in a training and test set. Through supervised learning, we aim to create models for quick and accurate predictions. Ultimately, this could streamline drug analysis in diverse lab environments.

2. Feature engineering

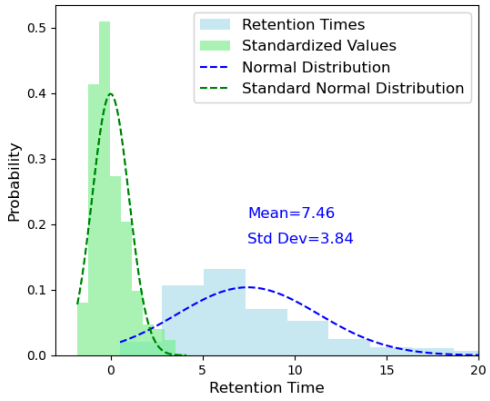

Figure 1: Distribution of standardized and unstandardized retention times

The provided training and test sets can be found on Kaggle. Both sets contain molecule details in SMILES format, along with ECFP and cddd data. Our first intuition was to add molecular proerties computed using the `rdkit` library, which can be found in `\Features\test-train_properties.csv`.

We then used one-hot encoding to differenciate the provenance of data based on the lab, which has an impact on the predictions as RT depends on the chromatography setup.

Basic feature enginnering was implemented including removal of constant predictors, remplacement of cddd's missing data using mean values and merging all features into a single dataset.

In the `importance.py` file, we used `RandomForestRegressor()` from `sklearn` to compute the order of importance of features. Not having satifactory results with `rdkit` properties and Lab features only, we later improved feature selection by using `SequentialFeatureSelector()` from `sklearn` in the `findMostImportantFeatures(Features='cddd', number_of_important_features=10)` function. Our findings demonstrate that the optimal features included Lab features, `rdkit` properties and the 100 most important cddd features. `selectFeatures()` function in `tools.py` allows easy extraction of the relevant features for a specific model among all the features engineeried data.

3. Linear Models

Linear Regression: Linear Regression model allows us to examine our engineered data features. We used the `LinearRegression()` class from `sklearn` and the `rdkit` properties. The first three features stood out as very important, so we tested using them alone and combined with ECFP and cddd. When we added ECFP and cddd, it caused a significant increase in errors. Our Kaggle scores[1] ranged between 6.89 (using only the top 3 features) and 13 (when ECFP and/or cddd was included). After the one-hot-encoding of the Lab and the correct selection of the features (100 best cddd), the error dropped to 1.125.

[1]Kaggle public scores are calculated using 27% of the final test data.

Ridge Regression: To determine if the Linear Regression's imprecision came from feature selection or the model itself, we applied regularization using Ridge Regression `Ridge()` from `sklearn.linear_model` and `GridSearchCV()` to tune the best regularization constant λ. With the optimal λ, the error decreased to 1.07.

Stochastic Gradient Descent: Stochastic Gradient Descent's implementation was executed using `SGDRegressor()` from `sklearn.linear_model`. The hyperparameters (α, η_0, penalty) were tuned using `GridSearchCV()`. With the best parameters being $\alpha = 0.0001$, $\eta_0 = 0.001$, penalty = `l2`, the error was similar to the Ridge one, which is coherent since `SGDRegressor(loss='squared_error', penalty='l2')` and Ridge solve the same optimization problem using different methods.

4. Non Linear Models : Neural Networks

Implementation using Pytorch: Neural Network's implementation began with the PyTorch [1] library. We first used basic layer layouts, along `ReLU` activation. A significant improvement was standardizing the input data before training, but this made error estimation less accurate. To avoid overfitting on the validation set, we implemented early stopping, resulting in a error's reduction of 0.16. Employing cross-validation via `KFold()` helped prevent overall overfitting. We tried to vizualise the effect of learning rate using `plotLR()`. Having way more hyperparameters to tune, we decided to transitioned to the Keras library due to easier tuning function implementation.

Implementation using Keras and Tensorflow : After prediting some `Pytorch` models, we switched to the `Keras` [2] library from `Tensorflow`. One of the reasons was to use the `keras.Callbacks`. This lets us use mechanisms as `EarlyStopping()` without a code implementation, but only by calling those callbacks when training the model. Another relevant callback we used was `ReduceLROnPlateau()`.

Hyperparameter tunining : By submitting numerous `Pytorch` and `Keras` models and not having any improvement, we started the hyperparameter tuning using `keras_tuner` library. We optimized the learning and rate decay, patience, loss functions as well as the layers architecture (number, size, activation functions) which led to an error reduction of 0.16.

The implementation of the cross-validation in Keras tuning was more difficult than expected, but was a critical asset for better error estimation. We defined a subclass of `keras_tuner.Tuner` called `CVTuner` to check on 3 different folds of the data the performance of each trial.

5. Discussion

Throughout the progression of our project, one guiding principle has been the scores displayed on Kaggle. This guidance increaseing the risk of overfitting to the 27% of the test set, to reduce this risk, we employ cross-validation for our Neural Networks and incorporate regularization into our linear methods.

To anticipate test and validation losses without requiring submission to Kaggle, we've integrated metric computations across all our files. However, during the implementation of non-linear models, the validation loss, intended to approximate the Kaggle score, underestimated the actual error to varying degrees across different models. This discrepancy complicates the enhancement of our model as it lacks a reliable indication of the approximation of the real error.

The project underscored the vital role of feature engineering. Initially overlooking Lab features, their inclusion notably reduced errors by approximately 5 in every models. Notably, the `SequentialFeatureSelector()` proved more effective in ranking the importance of cddd compared to `RandomForestRegressor()`. Despite employing SFS, introducing ECFP features failed to enhance predictions. The primary project limitation stemmed from inadequate methods for engineering ECFP properties, hindering their effective utilization.

Considering these limitations, the Neural Network created with Keras, using the laboratory features along with the top 100 cddd and `rdkit` properties, performs the best in terms of Kaggle's error.

Bibliography

[1] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library". [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[2] F. Chollet and others, "Keras: A Deep Learning Library for Tensorflow". [Online]. Available: https://keras.io/

[3] G. James, D. Witten, T. Hastie, R. Tibshirani, and J. Taylor, *An Introduction to Statistical Learning: with Applications in Python*. in Springer Texts in Statistics. Cham: Springer International Publishing, 2023. doi: 10.1007/978-3-031-38747-0.

[4] J. M. Brea, "Introduction to machine learning for bioengineers - Course Material". [Online]. Available: https://bio322.epfl.ch/