Shreyas Puducheri

Note: I assumed *prognosis* to be the column for the diagnosis/label. Not sure why it's called prognosis rather than diagnosis…

Data Exploration & Preprocessing

Ref: EDA & preprocessing/EDA.ipynb

The first thing I did was I visualized the feature and class distributions with bar plots, which can be found in the Jupyter notebook. Key takeaways were that the diagnosis distribution was uniform, and that most features were False. There were no missing features. Given these findings, no preprocessing was necessary, aside from label encoding the string labels, which was done later on during model training.

Then, I analyzed the relationship between features using both a Pearson's correlation matrix and a mutual information matrix. They both indicated the same thing – that there weren't clear relationships between the features, as most correlation coefficients and mutual information values were low. In fact, at this point I thought that the data looked "too clean" which is why I reached out to ask if it was a toy dataset – almost never would one find that patient symptoms are largely uncorrelated – health conditions, even when restricted to a specific body system almost always have indirect effects on a patient's state, and confounding factors like social determinants usually add to this.

I then calculated the mutual information between the features and each of the diagnoses, producing a k x n matrix of mutual information values (where there are k features and n diagnoses). This was visualized as a heatmap (which can be found in the jupyter notebook) and demonstrated that there were a few key variables that provided a high amount of shared information with the diagnosis. This told me that the classification task was probably going to be pretty easy – there are only a couple features that are relevant to each diagnosis label, and most features are only informative to only one diagnosis label. Again, probably something we'd never see in the real world.

Ref: EDA & preprocessing/train_test_split.py

I split the data into train and test sets (80:20) while stratifying by the label as to maintain a balanced distribution of testing and training cases. No other preprocessing was necessary

Model building and Training

Ref: Model/nb.py

Given my EDA insights, I decided to start something simple, like a naïve bayes model, since it's statistical in nature, easily explainable, and doesn't require hyperparameter tuning. In my code you'll see that I set up a 5-fold stratified CV, not for any particular reason other than just for thoroughness – I use the same setup for the other models, so that's my main reason for writing the code for that. Performance metrics were all perfect during cross-fold validation, confirming my initial predictions regarding the simplicity of the classification task.

Ref: Model/lr.py

I also trained a set of logistic regression models. I used an L1 regularizer, since the EDA informed me that a very limited number of features are relevant to each label, and the L1 penalty has the property of

shrinking coefficients to exactly zero, essentially acting as a feature selector. I selected the regularization parameter *C* on the basis of log loss over several iterations of stratified 5-fold cross validation. You can find a plot of the regularization path at *Model/regularization_path.png*.

Ref: Model/xgb.py

I also trained a XGBoost model for the multiclass classification task. To tune the hyperparameters, I again used stratified k-fold cross validation, but this time, I used a Bayesian grid search, implemented and logged using Weights and Biases (WandB). Since this is computationally intensive, I submitted it as a batch job to the BU Shared Computing Cluster (SCC) using the bash script *Model/xgb.sh.*

Ref: Model/nn.py

Finally, I trained a simple feed forward neural network with a single hidden layer. Honestly, its overkill at this point, given that all the previous, lightweight models achieved perfect performance.

I again used a stratified 5-fold cross validation with Bayesian grid search for hyperparameter optimization with WandB. I submitted the batch job to SCC with the script *Model/nn.sh* and used 1 GPU.

Testing/Validation

Ref: Testing/testing.ipynb

Testing the models was very straightforward. I just defined a function to create ROC and PR curves for each label, as well as to calculate accuracy, macro-F1 Score, Precision, and Recall. Every model had perfect performance on the test set so it was pretty redundant. A notable challenge here were some issues with the SimpleNN class I defined – I spent a while fixing bugs related to the prediction methods.

To mitigate overfitting, I consistently used 5-fold stratified cross-validation across all models. For logistic regression, I applied L1 regularization to prevent overfitting and perform feature selection. Also, by training multiple models (Naive Bayes, Logistic Regression, XGBoost, Neural Network), I could compare or combine them in an ensemble to potentially mitigate biases specific to any one model.

If i were to deploy any of these models it would probably be the naïve bayes model due to how lightweight it is and interpretable. Of course, before deployment it would be important to evaluate how fair the model is across race groups, socioeconomic groups, etc. Additionally, it would be important to retrain on real-world data (as long as IRB approval and patient consent is obtained) since research cohorts often differ from real-world cases.

Explainability

Ref: Explainability/explainability.ipynb

For naïve bayes, feature importance is pretty simple – since it uses bayes theorem, it has already computed the likelihood $P(X_i \mid y_n)$ where X_i represent the ith feature and y_n represents the nth label. Therefore, the log likelihoods represent feature importances. Inspecting the top 10 most important features for each label demonstrates that, as expected, there are 2-3 features for each label that are very important.

Similarly, for Logistic regression the feature importances with respect to each feature are pretty straightforward. Since I trained separate One-vs Rest logistic regression models for each label, I can just retrieve the feature coefficients from each model. The feature coefficient represents the effect that that feature has on model probabilities in the log-odds space. Therefore, the absolute value of the coefficient represents its importance. The results for this are more or less in line with that of naïve bayes.

For XGBoost, feature importance is calculated differently and globally across all labels, as label-specific importance isn't readily available in multi-class XGBoost. XGBoost builds a series of decision trees, each correcting errors of previous ones, using gradient descent to minimize loss. Feature importance is typically based on how much each feature contributes to decisions within these trees. The feature importance analysis for XGBoost wasn't that informative, since it wasn't specific to each label.

I did not conduct a feature importance analysis for the neural network due to time constraints. The neural network processes features through a hidden layer consisting of neurons, where weights are trained to map the preceding layer's neurons to the next layer's neurons. I would've conducted feature importance analysis with SHapley Additive exPlanations (SHAP), which calculates each input feature's contribution to the model's predictions at an instance level. I've done SHAP analysis on deep learning models for other works (see: https://www.nature.com/articles/s41591-024-03118-z/figures/6).

Ethics

It would've also been nice to conduct a subgroup analysis of performance, making sure that the model achieves high performance across demographic characteristics.