Introduction to Machine Learning

Final Project Deliverable - II Report

Option 1: End-to-End ML Model for Warfarin Dosage Prediction

Venkata Sai Polakam, Sai Charan Lenkallapally

Task 1: Data Preparation Strategy

- We have chosen two models namely Logistic Regression and Multi-Layer-Perceptron Classifier algorithms for working the Warfarin Dosage Prediction Problem.
- Initially, we've downloaded the dataset from PharmGKB website which had several columns. In that dataset, columns which will not be used for our analysis were deleted which led to a new dataset containing about 12-13 variables.
- Then, the dataset has been loaded into a Colab Notebook. Here, the first step we took is converting the categorical text data into numericals using Label Encoder.
- Now, we have checked for missing values and imputed the missing values with mean for numerical columns and the mode for categorical columns.
- We have also used Missingness Indicator Variable (MIV) to handle the missingness.
- Finally, we have separated the features variables from target Variable ("Therapeutic Dose of Warfarin") and they were saved as numpy variables.

Task 2: Python Code for Preprocessing

Here's the link for the Colab Notebook:
https://colab.research.google.com/drive/1yB816bIfg0vdTQxWCQQ5c7A2oKyx4AkP?usp=sharing

Task 3: Selection of ML Models and their usage

We have selected *Logistic Regression* and *Multi-Layer Perceptron Classifier* algorithms for working on this Warfarin Dosage Prediction Problem.

Here's our plan for utilizing them to make predictions in this problem:

Data Preprocessing:

Handle missing values: We can either remove rows with missing values or fill them in using a method like mean or median imputation. However we prefer to impute the missing values with mean for numerical variables and mode for categorical variables.

Convert categorical variables into numerical variables: Machine learning models require input to be numerical. Therefore, we shall convert categoricals into numerical using techniques like one-hot encoding.

Scale the data: Since MLPs are sensitive to the scale of the input data, we'll perform feature scaling using either of the Standard scaler or Min-Max scaler.
Split the Data: Splitting the data into a training set and a test set. The training set is used to train the model, while the test set is used to evaluate the model's performance on unseen data.

Model Training:
Logistic Regression: We'll use a library like scikit-learn in Python to train a Logistic Regression model on our training data.
MLP: We'll use a library like Keras or PyTorch to define and train an MLP. We need to decide on the architecture of the MLP (i.e., the number of layers and the number of neurons in each layer). We also need to choose an optimizer and a loss function. For a binary classification problem like this, binary cross entropy is a common choice for the loss function.

Model Evaluation:
Common metrics for classification problems such as accuracy, precision, recall, and the F1 score have to be calculated over testing set to evaluate the performance of the models we've built. We can also plot a confusion matrix to see how the model is performing on each class.

Hyperparameter Tuning:
If the performance of the models is not satisfactory, we can try tuning the hyperparameters. For Logistic Regression, we can tune the regularization strength. For the MLP, there are many hyperparameters we can tune, such as the learning rate, the number of layers, the number of neurons in each layer, etc.

Make Predictions:
Once we're satisfied with our models' performance, we can use them to make predictions on new data. Remember to preprocess the new data in the same way as wer training data.

Task 4: Choosing Optimal hyper-parameters
The most important part of building a model in machine learning is choosing the right set of hyperparameters. Here's what we do for that:

1.Grid Search: This method is quite simple. You just define the possible values for each of your hyperparameters, and grid search will train a model on every possible combination of these hyperparameters. It then selects the combination which performs best on the validation set.

2.Random Search: Random search tries a different strategy than grid search — instead of going through all combinations, it randomly picks sets of hyperparameters from within specified ranges for a fixed number of times/iterations. Random search can be better than grid search when you have many hyperparameters or large data sets because it's more efficient.

3.Bayesian Optimization: Bayesian optimization is an advanced technique where it treats the mapping from hyperparameter space to validation performance as a probabilistic function. After each evaluation round, this probability distribution gets updated so that our knowledge about good places in hyperspace increases over time.
Usually, there is a trade-off between model complexity and underfitting or overfitting when choosing hyperparameters.

Underfitting appears when the model becomes too basic to capture the data's hidden structure. This often happens if we set hyperparameters so as to impose very strict limits on the complexity of models (for instance, having very few layers or neurons in a neural network or not allowing a decision tree to split enough times).

Overfitting appears when the model becomes too complicated and starts capturing noise in data rather than its underlying structure. This often happens if we set hyperparameters so that models can be excessively complex (for example, having too many layers or neurons in a neural network or allowing a decision tree to split too many times).

To find this balance we may use such techniques as cross-validation for estimating how well does our model perform on unseen data; that allows us seeing whether it generalizes good enough to new observations and thus prevent overfitting. To avoid overfitting also one can apply regularization techniques which add penalty term into loss function for overly complex models.

Task 5: Model Evaluation

In a binary classification problem like this one, where we are predicting whether the required dose is high or low, there are several metrics that can be used to assess the performance of our models:
Accuracy: This is the simplest evaluation metric. It is the ratio of the number of correct predictions to the total number of predictions. While it's easy to understand, accuracy can be misleading if the classes are imbalanced.

Precision: Precision is the ratio of true positives (correctly predicted high dose) to the sum of true positives and false positives (low dose incorrectly predicted as high dose). Precision is a good metric to use when the cost of a false positive is high.

Recall (Sensitivity): Recall is the ratio of true positives to the sum of true positives and false negatives (high dose incorrectly predicted as low dose). Recall is a good metric to use when the cost of a false negative is high.

F1 Score: The F1 score is the harmonic mean of precision and recall. It tries to balance the two and is a good metric when we want to balance precision and recall and there is an uneven class distribution.
Area Under the ROC Curve (AUC-ROC): The receiver operating characteristic (ROC) curve is a graph of sensitivity (or recall) versus 1-specificity (or false positive rate) at different threshold values. It indicates how well a model can differentiate between classes. A single-value metric that shows how great is the ability of a model to distinguish between classes is called AUC-ROC. 0.5 means no discrimination, 1.0 means perfect discrimination and 0.0 means perfect misclassification.

Confusion Matrix: This table explains how well a classification model performs. True positives, true negatives, false positives and false negatives are recorded in this 2×2 matrix which enables us compute several metrics like precision, recall or F1 score.

The metric chosen depends on problem requirements. For instance, if there is a need for detecting high doses correctly even when some low doses are misclassified, we should maximize for recall. If both need to be balanced then choose F1 score.

Task 6: Mitigating Overfitting and Underfitting

We can tell if a model is underfitting or overfitting by checking how well it performs on the training set and a validation set:

Underfitting refers to a situation where the model does badly on both the training and validation data sets. This shows that the model is too simple and fails to capture the underlying structures in the data. In cases of underfitting, high errors will be recorded for both the training set and the validation set.

Overfitting occurs when a model does well on the training data but poorly on validation data. It means that such a model is too complicated since it fits noise from training examples rather than generalizing patterns. Consequently, overfitted models have low error rates for training samples while high ones for test samples.

We may draw learning curves by plotting against one another different measures of errors as we vary complexity levels like epochs in neural networks or depths in decision trees for this purpose of visualization.

If the model is underfitting, following strategies can be applied for mitigation:

Augment model complexity: For example, we can raise the quantity of layers or neurons in a neural network, or increase the maximum depth of decision tree.
Boost features count: We may invent fresh attributes or employ methods such as PCA to generate different combinations of characteristics.
If the model is overfitting, following strategies could be applied for mitigation:

Decrease model complexity: For example, we can reduce number of layers or neurons in a neural network, or decrease maximum depth of decision tree.
Regularization: This implies adding penalty to loss function for complex models. For instance L1 and L2 regularization phrases in loss function penalize large weights.
Dropout: During every training step random subset of neurons are dropped out i.e., deactivated. It helps prevent over-reliance on any one feature by the model.
Early stopping: Training has to be stopped when validation error starts increasing even if training error still decreases.

Task 7: Concept Map
Here's the link for our concept map:
https://app.diagrams.net/#G1jvl5Od6suJ7FLd_36D-fVPRU4ImpA_p_#%7B%22pageId%22%3A%22VUydFSxBYoSMAdlEaXDb%22%7D

Task 8:
8.1: Sample codes to train Multi-layer neural network models
Using Keras to Load Dataset:

```
import tensorflow as tf
fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]
```

Creating the model using Sequential API:

```
tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Input(shape=[28, 28]))
```

```
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Compiling the model:

```
model.compile(loss="sparse_categorical_crossentropy",
optimizer="sgd",
metrics=["accuracy"])
```

Training and Evaluating the model:

```
history = model.fit(X_train, y_train, epochs=30,
... validation_data=(X_valid, y_valid))
```

Using the model to make predictions:

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
```

8.2: Evaluation Metrics

Accuracy: Accuracy measures the proportion of correctly classified instances out of all instances.

Precision: Precision measures the proportion of true positive predictions out of all positive predictions, indicating the model's ability to avoid false positives.

Recall: Recall measures the proportion of true positive predictions out of all actual positives, showing the model's ability to find all positive instances.

F1-score: F1-score is the harmonic mean of precision and recall, providing a single metric that balances both precision and recall.

ROC score: ROC score, or Receiver Operating Characteristic score, evaluates the model's ability to discriminate between positive and negative classes across various thresholds, typically by measuring the area under the ROC curve.

Sample codes for calculating the above defined metrics:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
roc_auc_score, f1_score
y_pred = log_model.predict(small_X_train_flatten)
# Accuracy Score, precision score, recall score, f1 score
log_acc = accuracy_score(y_train, y_pred)
log_prec = precision_score(y_train, y_pred, average='macro')
log_recall = recall_score(y_train, y_pred, average='macro')
```

log_roc = roc_score(y_train, y_pred, average=’macro’)
log_f1 = f1_score(y_train, y_pred, average='macro')

Task 9:
9.1:
These are brief explanations of the four hyperparameters that we chose for multi-class classification using KNeighborsClassifier from sklearn:
n_neighbors: It tells us how many nearest neighbors should be taken into consideration while predicting. In a multi-class classification algorithm, n_neighbors indicates the number of training samples around the new data point that will vote for its class.
metric: This is used to find out similarity between two instances based on distance measure. For multi-classification task, we must select appropriate metric since it affects calculation of distances between objects belonging to different classes.
p: The power parameter in Minkowski metric; when p=1 it corresponds with Manhattan distance and when p=2 (default) it means Euclidean distance. However, if we have multi-class problems with high-dimensional data then higher values of p can be chosen.
n_jobs: It allows you to use multiple cores or CPUs simultaneously for computation. You can set -1 so that all available CPU cores are utilized which will speed up the process especially when dealing with large datasets in multi class classification problems.
9.2: Sample training code for KNN Model

```
from sklearn.neighbors import KNeighborsClassifier
knn_model = KNeighborsClassifier(n_neighbors = 6, metric = 'euclidean')
knn_model.fit(X_train, y_train )
knn_model.predict(X_valid)
```

Task 10: Review Logistic Regression
10.1:
Q1: Why does the logistic regression can be used for binary classification?
The logistic regression can be used in binary categorization due to the fact that it predicts the chances of an instance falling under either of the two groups. The logistic function takes any real number and returns a value between 0 and 1, which can be understood as the estimated probability that the instance is positive (or negative). Logistic regression does this by dividing all points into two classes according to their predicted probabilities based on this function.

Q2: What's the cost function for logistic regression to implement the binary classification?

The cost function of log loss or logistic loss is used for binary classification logistic regression. It punishes deviations between prognoses made by the model from correct class label being either 0 or 1. In order to achieve accurate binary classification after applying a threshold, we should find such values of parameters that make forecasted probabilities close enough to actual labels on training set thus minimizing this log loss over all examples.
The log loss is calculated as:
Log Loss = -[y*log(p) + (1-y)*log(1-p)]

10.2: Sample training code for Logistic Regression Model

```
from sklearn.linear_model import LogisticRegression
logistic_model = LogisticRegression(penalty = 'l2', C = 1, random_state = 0)
logistic_model.fit(X_train, y_train)
y_logistic_pred = logistic_model.predict(X_train)
```

Task 11: Support Vector Machine

11.1

Q1: What's the objective of SVM to find the optimal linear classifier?

Support Vector Machines (SVMs) are designed to create lines or boundaries that separate various groups in datasets with the largest possible spaces between them which is also called a margin. This helps the classifier understand new data better. However, there may be instances where the data is not separable linearly or when outliers exist. In such cases, SVM allows for some misclassifications through a flexibility parameter referred to as C. C determines the extent of allowable errors; thus, one looks for an optimal boundary that differentiates classes most while minimizing mistakes made by it against them because this tradeoff enables our model to generalize well on unseen points

Q2: What are available kernel functions for SVM to conduct linear and non-linear classification?

The common kernel functions available for SVMs to perform linear and non-linear classification are:

Linear kernel: It is used for linear classification problems.

Polynomial kernel: Useful for modeling non-linear relationships by adding higher-order polynomial terms.

Gaussian RBF (Radial Basis Function) kernel: It is highly effective for non-linear problems by mapping data to an infinite-dimensional space.

Sigmoid kernel: Can be useful for some non-linear problems, though it violates certain mathematical conditions.

11.2: Sample Training Code for SVM model

```
from sklearn.svm import SVC
# define linear kernel using polynomial features
polynomial_svm_clf = make_pipeline( PolynomialFeatures(degree=3),
StandardScaler(), LinearSVC(C=10, max_iter=10_000, random_state=42))
polynomial_svm_clf.fit(X, y)

# define polynomial kernel
from sklearn.svm import SVC
poly_kernel_svm_clf = make_pipeline(StandardScaler(),
SVC(kernel="poly", degree=3, coef0=1, C=5))
poly_kernel_svm_clf.fit(X, y)
# define Gaussian RBF kernel
rbf_kernel_svm_clf = make_pipeline(StandardScaler(),
SVC(kernel="rbf", gamma=5, C=0.001))
rbf_kernel_svm_clf.fit(X, y)
```

Task 12: Decision Tree

12.1:

Q1: Describe how the decision tree is constructed. How is each feature selected as a single node?

The decision tree is built by means of the CART (Classification and Regression Trees) algorithm, which follows these steps:

1. The algorithm commences at the root node with all of the training data.

2. To achieve purest sub-nodes (that is to say, nodes having minimum impurity), it assesses each feature's every conceivable split point.

3. The current node is chosen as the one that results in maximum reduction of impurity over all features and their split points.

4. Two child nodes are created from this selected feature and split point by parting data accordingly.

5. This procedure is repeated on each child node successively until some stop criterion is satisfied such as maximum depth reached or node becomes pure etcetera

Q2: What are hyper-parameters available for decision tree? How to avoid overfitting in decision tree?

The main hyperparameters for decision trees in scikit-learn are:

1) max_depth - The maximum depth of the tree. Setting this will help in preventing overfitting.

2) min_samples_split - Minimum number of samples required to split an internal node.

3) min_samples_leaf - Minimum number of samples required to be at a leaf node.

4) max_features - Maximum number of features to consider when looking for the best split.

5) max_leaf_nodes - Maximum number of leaf nodes allowed.

In order to restrict overfitting in decision trees, we regularize them using these hyperparameters. In other words,

- To limit the depth and complexity, reduce max_depth
- Increase min_samples_split and min_samples_leaf so that they do not grow on small number of samples, this will also make the tree less complex
- When we set max_features to be a smaller value it makes that at each split, fewer features need consideration
- We can also cap total number of leaf nodes by setting max_leaf_nodes

Regularization means tuning these hyperparameters which restricts freedom necessary for overfitting such as depth, splits, leaves etcetera.

12.2: Sample Training Code for Decision Tree Model

```
from sklearn.tree import DecisionTreeClassifier

tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42)

tree_clf.fit(X_iris, y_iris)

tree_clf.predict_proba([[5, 1.5]]).round(3)
```

```
tree_clf.predict([[5, 1.5]])
```

Task 13: Random Forest

13.1:

Q1: Describe how the ensemble learning works in random forest?

Random forests use the bagging learning technique called an ensemble. Here is a simple explanation of how it works:

1. The random forest algorithm creates many decision trees from the training data.
2. It picks a random bootstrap sample for each tree out of the original training set (sampling with replacement).
3. Then, it trains every decision tree on the bootstrapped sample with an introduction to randomness.
4. When predicting on new data points, all individual decision trees' predictions in the random forest are combined together.
5. The predictions are aggregated by taking mode for classification problems (the most commonly predicted class) or mean for regression problems.

Q2: Describe the concepts behind bagging and random patches/subspaces?

Here's a brief explanation of bagging and random patches/subspaces:

Bagging:

- Stands for bootstrap aggregating
- Each model in the ensemble is trained on a random bootstrap sample of the training data (sampled with replacement)
- This introduces randomness and makes the models differentiates from each other
- It helps reduce variance and overfitting compared to a single model

Random Patches:

- In addition to bagging training instances, it also samples random subsets of features for each model

- So each model sees a random patch of the training data (random instances and random features)
- This further increases randomization and decorrelation between the models

Random Subspaces:

- A special case where all training instances are used (no bagging), but features are randomly sampled for each model
- The models see random subspaces of the full feature space
- Again increases randomness across models to reduce variance

Q3: How does the feature importance work in Random Forest?

Random forests can measure the relative importance of each feature in a straightforward way:

1. For each feature, it looks at how much on average the tree nodes that use that feature reduce impurity (e.g. gini impurity for classification) across all trees in the forest.
2. It calculates a weighted average, where each node's weight is proportional to the number of training samples associated with that node.
3. The final feature importance scores are scaled so that they sum up to 1.
4. These scores are automatically computed during training and stored in the feature_importances_ attribute of the RandomForestClassifier/Regressor object.

13.2: Sample Training Code for Random Forest Model

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1, random_state=42)

rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

Task 14: Sample Codes for Hyper Parameter Tuning

```
from sklearn.neighbors import KNeighborsClassifier

KNN_classifier = KNeighborsClassifier()

# parameter 1: n_neighbors

n_neighbors_list = [1, 3, 5, 7, 9]

# parameter 2: weighting strategies

metrics_list = ["uniform", "distance"]

# Define the hyper-parameter combination

KNN_param_grid = {

'n_neighbors': n_neighbors_list,

'weights': metrics_list

}

#define grid_search

from sklearn.model_selection import GridSearchCV

grid_search = GridSearchCV(estimator = KNN_classifier, param_grid =
KNN_param_grid, cv = 10, scoring = 'accuracy', return_train_score = True)

# fit grid-searchcv on training data

grid_search.fit(X = small_X_train_flatten, y = small_y_train)

# get best parameters

grid_search.best_params_

# get best estimator

best_knn_model = grid_search.best_estimator_

best_knn_model
```