SML Individual Project

Algorithm Benchmarks

Author: *Dimitri Kestenbaum*

Abstract

Within this document I will summarize conceptually five different machine learning models: `linear regression`,`logistic regression`,`decision tree classifier`,`random forest classifier`, and `SVM`. I will benchmark the application of each model in Python. I will use the `sklearn` machine learning library for the majority of the application sections. The data set used for all benchmark applications included in this report is from a Taiwanese banking institution who'd like to predict a client's likelihood of defaulting on their credit card using as many as 23 independent variables.

All the mathematical equations in this file have been manually created by myself using a LaTex equation editor.

Multivariate Linear Regression

`Linear Regression` is a statistical learning algorithm used within a supervised setting which can be used for predicting a continuous numerical value. Linear regression can be split into two different types: `univariate` or simple regression and `multivariate` regression. What differentiates these two variations of the algorithm are whether it's fit to multiple independent variables for prediction. `Univariate` models use only one independent variable as a predictor, while `multivariate` models use multiple to estimate a prediction. This algorithm predates its modern machine learning application significantly, as it is considered one of statistics most fundamental mechanisms.

In a nutshell, linear regression fits a line or hyperplane to the data. This regression line represents the best fit for measuring a constant rate of change between a dependent variable or `response variable` and one or multiple independent variables or `features`. The slope of the regression line is dependent on coefficients or weights which are independently assigned to the features within the model. For example, if our objective was to predict the temperature on a given day, one feature which may intuitively seem like a candidate for a high weight is the time of the year (season). As we can assume that the season most likely has a high correlation with the response variable or in other words is an important factor in deriving its value mathematically. It is important to consider that for linear regression to produce accurate predictions, some assumptions must be met. Here are the two most important assumptions that should be met:

- The data should follow a linear pattern.
- The features should not be too correlated with each other (`multicollinearity`).

Equation

The equation of multiple linear regression is certainly one most people have encountered in their grade school math curriculum, as it is simply the equation for slope, or change of a dependent variable as a ratio of change in an independent variables. The equation is as follows:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 ... + \beta_p x_p$$

y represents the response variable, *x* the independent variable, and beta zero through beta p indicate the intercept coefficient and the coefficients for the features.

Linear Regression's Objective Function

In order to fit a line that best matches the pattern of the data, linear regression, just as all other learning algorithms do, depends on an objective or cost function which acts as a feedback loop to the algorithm indicating how good of a job it is doing and by how much. By doing this, the algorithm can iteratively adjust its coefficients to minimize the output of the objective function and subsequently improve its fit. One common objective function used is RSS or residual sum of squares. Which penalizes predictions with large residuals by squaring their value. A residual is the difference between a prediction value outputted by the model and an actual value which we figuratively keep out of the model's site until it's time to evaluate its performance. The RSS is than just a sum of the residuals for all predictions. The equation looks as follows:

$$RSS = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Here *yi* represents an actual label value and *f(xi)* a predicted or estimated value. When we subtract predicted value from the actual value we get the residual. We then square the residual and perform summation across all predicted values to find the RSS.

Pros and Cons of Linear Regression

Here are some points for what has made this algorithm stay in the mix for so long as well as some drawbacks that might make it not the best tool for a particular regression task.

The Pros:

- Its interpretability and easy-to-grasp mathematical form make it a popular go to for baseline modeling and pedagogy.
- Performs very well on data which does follow a linear pattern.
- There are extensions and variations of linear regression, such as interaction and polynomial terms and ridge and lasso regression which can extend the models flexibility and mitigate some of its other drawbacks.
- Compared to many other models, linear regression is very computationally efficient due to its straightforward algorithm.

The Cons:

- One major drawback to linear regression is that it assumes independence of the features. This is the same issue as multicollinearity. This means applying linear regression requires very careful feature selection.
- The relationship between a response variable and its feature is somewhat rarely a linear relationship meaning that added complexity might need to be added in the form of polynomial terms, etc.
- Linear regression is very sensitive to outliers which can severely skew predictions.

Applying Linear Regression in Python

To create a benchmark demonstration of applying linear regression using Python, I worked with both the `scikit learn` and `statsmodels` modules. While typically this data set is used for classification, in this application I will predict the value of `LIMIT_BAL`, which represents the amount of total credit provided to a client at the household level, using a selection of the other features. I selected my variables based on two important criteria: the `Pvalue` of a feature, as well as whether or not a feature had a high level of correlation with other features (`multicollinearity`).

Next, I fit the data to the statsmodels `OLS` class in order to view a summary output of the coefficients and their Pvalue's. OLS which stands for ordinary least squares, is an implementation of linear regression which uses RSS to optimize the fit of the regression line. The summary indicated the following:

Thanks to the detailed `summary()` method from statsmodels OLS class we have validated the statistical significance of the included features and can be confident that these features are related to the response variable. Fitting and predicting using scikit learn's linear regression class on a train/test split of the model results in an RSS of `67958479153053.086`, see the code for this finale prediction below:

```
X_train, X_test, y_train, y_test =
train_test_split(X,y,test_size=0.3,random_state=123)

lr = LinearRegression()

lr.fit(X_train,y_train)

y_preds = lr.predict(X_test)
```

Logistic Regression

`Logistic regression` is also a classical statistical learning model which offers an easily interpretable algorithm, but instead for classification tasks. The most common application of logistic regression is binary classification where the objective is to predict the probability of a certain event occurring (target variable) given a set of features. The event of interest can be represented either by a discrete binary label or a probability ranging from 0 to 1. Often the probability value is discretized by using a probability threshold (typically 0.5) where any probability equal to or greater than the threshold is considered a positive response label (typically 1).

While logistic regression is optimal for binary classification, it is possible to extend the model to support multi-class classification. This can be accomplished by dividing the multi-class classification dataset into multiple binary classification datasets and then fitting logistic regression to each one. The means by which this is applied can be divided into two different strategies:

- `OvA`: In One-vs-All, we create as many binary classification problems as there are classes of interest. The problems attempt to classify whether an observation belongs to one isolated class or a grouping of all the other class values. For example, if our multi-class classification problem consisted of classifying an observation as A, B, or C, we could do the following:

- A vs (B, C)
- B vs (A, C)
- C vs (A, B)

Whichever of the above binary classification problems results in the highest probability, we select as the observations class.

- `OvO`: In One-vs-One, we create a binary classification for every pair of classes. For example, if we continue with the previous example of a multi-class classification problem between A, B, or C, then OvO would look as follows:
- A vs B
- A vs C
- C vs B

Then a voting process occurs where the majority classification is considered the output of the OvO strategy.

Equation

To make predictions ranging from 0 and 1 to model probability of the event of interest occurring, we need a function which outputs such values. The aforementioned function is called the sigmoid function and is the heart of logistic regression. Its formula is as follows:

$$S(x) = \frac{1}{1 + e^{-x}}$$

Using this function, we can plug in the linear combination of coefficients and corresponding feature vectors into *x*, hence the *regression* in logistic regression.

Logistic Regression's Objective Function:

The beta coefficients must be estimated so that logistic regression's output predicts probabilities of the likelihood of an observation to fall in the positive response class which are as close as possible to the observed labels (0 or 1). In the case of this data set, the event of interest is credit card default in the following month (`default.payment.next.month`). The mathematical function which can be maximized to determine these optimal coefficient values is called the `likelihood function` formulated as follows:

$$logLikelihood(\beta) = -\sum_{j=1}^{M} y_j log(\widehat{y_j})$$

We optimize the likelihood function above by finding the coefficients that minimize the function. These will be the estimated beta coefficients we plug into the *x* within the sigmoid function. Alternatively, we can use maximum likelihood and maximize it to find the optimal beta coefficients.

Pros and Cons of Logistic Regression:

The Pros:

- Similar to linear regression, logistic regression is a relatively easy model to implement and interpret.
- It can be extended to handle multi-class classification using OvO and OvA strategies.
- Performs well on simple datasets that are clearly separated.
- Makes no assumptions about the underlying distributions of the classes.

The Cons:

- Much like its linear counterpart linear regression, logistic regression assumes linearity between the dependent and independent variables which can lead to added bias for non-linear relationships.
- While it can use extensions to handle multi-class classification, many related extension techniques are not efficient.
- Logistic regression is somewhat sensitive to multicollinearity.

Applying Logistic Regression in Python

To apply logistic regression in Python, I used `sklearn`'s `LogisticRegression` class. In terms of preprocessing, I first established the feature matrix `X` and target variable vector `y`. I'll use `default.payment.next.month` as the target for all subsequent benchmarks including this one. Nextly, I use a forward stepwise feature selection function from the `mlxtend` module. Forward stepwise feature selection entails starting with a null model (no predictors) and then sequentially adding features one-at-a-time which lead to a better model. I load the `LogisticRegression()` and configure the forward stepwise feature selection class as follows:

```
#initiate sklearn's logistic regression
log_reg_mod = LogisticRegression() #class_weight = 'balanced'
for_step_select = sfs(log_reg_mod, k_features=8, forward=True, verbose=2,
scoring='accuracy')
```

`k_features = 8` indicates I want FSFS to select only a subset of eight features to keep for the model.

The selected features include: `['EDUCATION', 'MARRIAGE', 'PAY_0', 'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6']`.

Finally, I perform a `train/test` split, train the model, make predictions, and evaluate model performance as follows. Ultimately, the model provides a mediocre AUC of `0.6042`. The code to accomplish all of this is as follows:

```
log_reg_mod.fit(X_train,y_train)

y_preds = log_reg_mod.predict(X_test)

fpr, tpr, thresholds = metrics.roc_curve(y_test, y_preds, pos_label=1)

metrics.auc(fpr, tpr)
```

Decision Tree Classifier:

A `decision tree` is a popular algorithmic method for dividing a feature space into discrete partitions or regions through a series of conditional branches (decisions) which ultimately allow us to derive a prediction for both regression and classification use cases. Tree-based models are extremely popular in machine learning both for their effectiveness and their interpretability. Parallels are often drawn that tree-based models imitate the process of human decision making. Here we'll only consider a tree classifier as it is more relevant to the credit card default dataset. The only large contrast between regression and classification trees is in how predictions are determined from a subregion of the feature space. In classification, the predicted label is that of the majority class within the segmented region of the feature space.

To understand more closely how this algorithm works, a quick description of a decision trees structure is important. A tree can be thought of is a series of consecutive logical splits. These splits are termed branches or internal nodes. Each split consists of a logical expression with discrete binary response possibilities. For example, within the context of the credit card default dataset one such branch could consist of a conditional statement like so `LIMIT_BAL >= 1000`. For all observations for which this statement is `True`, they would continue down left branch. For those which don't meet this criterion (`False`), they go down the right branch. A node in the tree that is at a dead end, to say it has no subsequent branches is considered a terminal or leaf node. These nodes are what allow us to classify an observation. If it falls into a leaf node in which the positive class (1) is the most prevalent than the algorithm will classify it as belonging to the positive class. Just like all machine learning models the decision tree is motivated using a cost function. The criteria for the number of splits in a tree is contingent or whether another split would result in better predictions (an improved objective function for each leaf node). Another important distinction to make is the order in which branches occur. The algorithm scans through the feature space and determines what splits best divide the data into the most homogeneous regions possible. Complete homogeneity would mean only one class appearing in that region which would mean it is possible to make an accurate prediction. The term for this homogeneity within leaf nodes is node purity.

Often decision trees tend to become overly fit to the specific complexities of a training set. When this happens, many branches will be created. This is often a symptom of an overfit model. One way to counter act this with a decision tree is by either setting a parameter to limit the maximum number of branches allowed, or by systematically removing branches in the order in which removals cause the least increase in error.

Decision Tree Classifier's Objective Function:

Three different measures can be used as a decision tree classifier's objective function. The first of which is `classification error rate`. This measure is very easy to compute and is defined as follows:

Here error is determined by subtracting the proportion of observations of the majority class to all other observation within a terminal node from one. Intuitively the higher the proportion of observations of the majority class the smaller the error. However, the drawback to using classification error rate becomes clear when we have a large decision tree with many branches. Many branches result in leaf nodes with smaller number of observations. Imagine there're only two observations within a leaf node, in this case the classification error rate can only ever be 0 or 50.

The two alternatives to classification error rate are `the Gini index` and `binary cross entropy` or log loss for short. The Gini index works like the classification error rate but instead multiples the proportion of each class by one minus this proportion and then sums this across all different classes. The Gini index comes from econometrics and is usually used to represent wealth inequality within countries where a value of one denotes complete inequality and zero perfect equality. In this context a Gini index of one indicates an extremely not pure node and a Gini index of zero indicates a completely pure node. The equation for the Gini index is as follows:

Log loss is the inverse logit function and can be used to optimize logistic regression. It takes on a value of zero when nodes are completely pure and one when they're completely impure just like the Gini index. It's defined like so:

Pros and Cons of Decision Tree Classifiers:

The Pros:

- Highly interpretable and easy to conceptualize.
- They're a great baseline model as we can deduce things like feature importance by applying them.
- Very minimal data preprocessing required (depending on software libraries).
- Non-parametric.

The Cons:

- Decision trees are highly prone to overfitting training data causing large variances in test error.
- More computationally intensive than the previous models.
- Decision trees have several hyperparameters and often require resource intensive tuning.

Applying Decision Tree Classifiers in Python

I used `Sklearn`'s `DecisionTreeClassifier` class to apply this model in Python. Forward stepwise feature selection was implemented again for feature selection.

```
clf = DecisionTreeClassifier(random_state=0)

for_step_select_clf = sfs(clf, k_features=20, forward=True, verbose=2,
scoring='accuracy')
#fit the forward stepwise selection class to the data
for_step_select_clf = for_step_select_clf.fit(X, y)
```

Notice the different choice in the `k_features` argument as opposed to fitting the logistic regression model to the `sfs` function. Considering decision trees can handle a large number of features, this was accounted for by increasing the number of best features FSFS should return.

After feature selection, the data must be split once again into train and test sets. While the previous models do not have any critical hyperparameters to tune, tree-based models do. Using `Sklearn`'s `GridSearchCV` this can be accomplished like so:

```
#create params dictionary
param_dict = {
    'criterion':['gini','entropy'],
    "max_depth": range(1,5),
    "min_samples_split": range(1,5),
    "min_samples_leaf": range(1,5)
}

grid = GridSearchCV(clf,
                    param_grid=param_dict,
                    cv=5,
                    verbose=1,
                    n_jobs=-1)
grid.fit(X_train,y_train)
```

Firstly, we create a parameter dictionary mapping every hyperparameter we intend on tuning with values for grid search to search over. Grid search then iteratively tries all possible combinations of hyperparameter values, while keeping track on how they impact model performance on the trainset. Here we have `128` candidate value combinations and given that we specify that we want `cv = 5`, meaning five-fold cross validation this results in `640 fits`. The hyperparameters used to tune the decision tree classifier:

- `Criterion`: The function used to measure how good a split is.
- `Max_depth`: The maximum number of branch levels a tree can have.
- `Min_samples_split`: The minimum number of samples necessary to create a branch.
- `Min_samples_leaf`: The minimum number of samples to qualify as a terminal or leaf node.

The best parameters returned by GridSearchCV are:

```
{'criterion': 'gini', 'max_depth': 1, 'min_samples_leaf': 1, 'min_samples_split':
2}
```

Next, I fit the model using these parameters and evaluate the model using `AUC`. With tuned parameters the decision tree classifier performs pretty well with an AUC of `0.64`.

Random Forest Classifier

Random forests are an example of an `ensemble` learning method meaning they use a collection of learning algorithms in making predictions. The random forest as its name suggests is an ensemble of decision trees. The algorithm instantiates a large number (commonly between 100-500) different decision trees and then uses all their individual predicted classes to tally a vote. The class with the most votes then becomes the model's prediction. The idea here is that while decision trees are highly sensitive to the specific patterns within a trainset, if we instead use many trees with randomly sampled training observations as well as shuffled predictors to ensure different feature importance’s than we can mitigate this unwanted overfitting.

The brilliant aspect about how a decision tree works is that it invokes a sort of diversification of results to account for the variance of the individual trees (estimators). This concept is what truly revolutionized tree-based approaches, by mitigating the overfitting drawbacks associated with a decision tree. However, the added complexity of a random forest as compared to a decision does come at a cost. The primary issue is computation time, considering each individual estimator is a decision tree. Additionally, due to the increased number of moving pieces and required considerations, random forest has some important hyperparameter that need to be tuned properly.

Random Forest Classifier's Objective Function:

The objective function for a random forest classifier is the same as that for a decision tree classifier and must be optimized at the node level. The two objective functions or criterions are the Gini index and entropy as previously explained in the decision tree classifier section.

Pros and Cons of Random Forest Classifiers:

The Pros:

- Robust to variances in training data (avoids overfitting).
- Just like the decision tree classifier, random forests allow you to see feature importance's explicitly.
- Very minimal data preprocessing required (depending on software libraries).

The Cons:

- Very computationally intensive and can run very slow with many estimators.
- Random forests lose a bit of the interpretability of simple decision trees in their large-scale complexity.
- Lots of thought must be given to optimizing the hyperparameters.

Applying Random Forest Classifiers in Python

To benchmark a random forest classifier I import the `RandomForestClassifier` class once again from `sklearn`. I use FSFS again to select the best features, this time setting `k_features = 20`. Considering the algorithm will randomly subsample the data within the many decision trees used in the forest. Therefore, it makes sense to give the model more features to sample from. FSFS determines the following features the best predictors for the model: `['cust_id', 'LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE', 'AGE', 'PAY_0', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6', 'BILL_AMT1', 'BILL_AMT2', 'BILL_AMT4', 'PAY_AMT1', 'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT5', 'PAY_AMT6']`.

I apply hyperparameter tuning once again as random forest has many important hyperparameters to tune. Tuning a random forest can make a very significant difference in model performance. I chose to utilize the `RandomizedSarchCV` as opposed to the `GridSearchCV` to speed up the tuning process. The randomized search works almost the same but chooses parameter combinations randomly. Additionally, the randomized grid search allows you to select the number parameter combinations which are sampled via the `n_iter` parameter, allowing you to virtually decide between run time and quality of the searching procedure. My code for this grid search is as follows:

```
#create params dictionary
rf = RandomForestClassifier()
#create parameter dictionary
param_grid = {'bootstrap': [True, False],
              'max_depth': [10, 20, 30, 40, 50, 60, 70, None],
              'max_features': ['auto', 'sqrt'],
              'min_samples_leaf': [1, 2, 4],
              'min_samples_split': [2, 5, 10],
                'n_estimators': [130, 180, 230,1000]}
#initiate grid search
grid = RandomizedSearchCV(rf,param_grid,
                          n_iter=50,
                          random_state=123,
                          verbose=3)
#fit train data
grid.fit(X_train,y_train)
```

The new parameters featured here that are important are as follows:

- `bootstrap`: Whether to use bootstrapped sampling when building the estimators (trees).
- `max_features`: The number of features to consider using when searching for the best split.
- `n_estimators`: The number of trees in the random forest.

Finally, I fit the model with the parameters deemed optimal by the randomized grid search and make predictions. This model performs the best so far with an AUC of `0.657`.

Support Vector Machines:

In simplest terms an `SVM` is a statistical learning model which fits a hyperplane within feature space to classify data points discretely. While other versions of the SVM model exist for regression uses cases, such as the support vector regressor, I'll focus solely on the classical usage of the SVM model. To explain the core mechanism behind SVM, I'll focus on a simple example. Imagine you have a dataset that consists of two discrete classes. One class could be circles and the other could be squares. Now, imagine these datapoints are scattered across feature space and you want to fit a line or hyperplane to create a linear separation boundary of the circles and squares. That way you can simply classify all the points that fall above the line as circles and all that fall below as squares. The issue here arises from infinite possibilities in space. There's no easy way to determine which is the optimal position for this line to be in, because you could slightly change its position infinitely. There needs to be a more concrete set of rules for this algorithm to be computationally efficient and just generally effective at its task.

The answer to this dilemma is the margin. The margin is essentially the maximum space between the hyperplane and its closest data point(s) in both directions. The datapoints which have the minimum distance from the hyperplane therefore define the margins border and are therefore termed `support vectors`. The reason we want to maximize this margin is because the larger it is, the more confident we can be in the accuracy of our classifications based on which side of the hyperplane data points are on.

The next limitation that arises from this point is what happens when the data is not linearly separable, meaning there's no clearly defined hyperplane to separate classes without there being misclassifications. One way of making the algorithm less strict when it comes to linear separation boundaries is adding a slack variable to the objective function (described below) to correct the objective function so that it satisfies its constraints. This concept is called a hard/soft margin. A hard margin as its name implies is strict and allows for no slack variables, while a soft margin allows for the use of slack variables to make the decision boundary less strict but imposes bias by making the fit less accurate to the actual data. The use of slack variable is controlled by a `C` variable which represents the budget for adding slack variables. If C is very low, larger slack variables will be used while when the opposite is true, slack variables will be more highly penalized and will then shrink. This penalty term concept is similar to the ideas behind `L2` regularization.

Another interesting way to solve more extreme situations, where data is totally not linearly separable is `SVM kernels`. A kernel in this context is an interesting mathematical hack used to make two-dimensional data, that's not linearly separable in two dimensions linearly separable in higher dimensional space by engineering new variables to add new dimensions to the data, then finding the linear separation hyperplane in three-dimensional space and finally projecting it back down to 2D space.

I won't delve too deep into the mathematics here, but I will enumerate and summarize the two most important kernels that can be applied to SVM from a conceptual standpoint.

- `Linear kernel`: this kernel is to be used when the data is linearly separable. This kernel doesn't project onto higher dimensional space. It is given by the inner product of two vectors.

- `Polynomial kernel`: This kernel can be used when the data is not linearly separable and adds a polynomial term much like the polynomial term in regression. The trick here is still that the kernel allows for the inner product of vectors to be used as opposed to the actual individual data points. However, it does add many added dimensions to the data which can result in very long run times. Additionally, the polynomial regression introduces two new hyperparameters: `Degree`, which indicates the degree of the kernel's polynomial term, (higher degree, more flexibility) and `Gamma`.

- `Radial kernel`: Based on the radial basis function (`RBF`). This kernel can be effective when dealing with non-linearly separable data. Interestingly, RBF has the property of infinitely expanding the dimensions of the data. This makes it especially useful for non-linear geometrical patterns once projected back down to the original dimensions.

SVM can also be extended to multi-class classification problems using `OvO` and `OvA`. This extension beyond the binary-class setting is methodologically identical to OvO and OvA with logistic regression.

SVM's Objective Function:

image

Here we want to maximize the sum of the inner term because when a label is correctly classified it results in one. However, we want to minimize the penalty term. The lambda symbol is a tuning parameter to control the regularization of the coefficients. A small lambda value corresponds to a small C value, meaning the model will allow a lot of violation (soft margin). The equation can be expressed in many forms including the hinge-loss function, but for the purpose of this project, I will not go any further with its explanation.

Pros and Cons of the SVM:

The Pros:

- Kernelization makes SVM versatile and effective in high dimensional spaces.
- Works efficiently and accurately on linearly separated data.
- Gives the user a lot of control over its functionality.

The Cons:

- Very slow run time on large datasets, often making it unsuitable for a particular problem.
- When data is not linearly separable SVM requires a lot of careful tuning.

Applying SVM in Python

For the application of SVM in Python, I continue utilizing FSFS for selecting features. I keep the `k_features = 20` argument the same considering SVM is known for handling high dimensionality. The `sfs` function outputs the following features: `['cust_id', 'LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE', 'AGE', 'PAY_0', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6', 'BILL_AMT1', 'BILL_AMT2', 'BILL_AMT4', 'PAY_AMT1', 'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT5', 'PAY_AMT6']`.

I use `sklearn`'s `LinearSVC` class. This model is like the `SVC` model however has the property of working more efficiently with large datasets. It only uses the linear kernel and allows for some additional hyperparameter options. One important preprocessing step I applied is scaling the features. Since SVM deals with distances we need to convert all the different dimensions to the same scale. I then perform a train/test split like in all previous applications. Next, I initialize a grid search to tune SVM's many parameters once again using the randomized grid search. The parameter grid looks like this:

```
# defining parameter range
SVM_clf = LinearSVC()
#parameter dictionary object
param_grid = { 'C':[0.1,1,100,1000],
              'loss':['hinge','squared_hinge'],
              'class_weight':[None,'balanced'],
              'max_iter':[500,1000,2000]}
#grid search
grid = RandomizedSearchCV(SVM_clf,param_grid,verbose=3)
grid.fit(X_train,y_train)
```

I'll once again map the parameters used in the grid search with their corresponding significance:

- `C`: The regularization penalty term. It controls the budget for misclassifications the SVC will use.
- `loss`: Determines whether to use a hinge or squared-hinge loss function as the criterion or objective function.
- `class_weight`: Takes a value of `balanced` or None and determines whether balance weights according to class ratio or incidence rate.
- `max_iter`: How many iterations of the model to run at maximum.

I then fit the model as follows:

```
SVM_clf = LinearSVC(C=1000, class_weight='balanced', loss='hinge', max_iter=6000,
random_state=123)
#fit
SVM_clf.fit(X_train, y_train)
#predict
y_preds = SVM_clf.predict(X_test)
```

```
fpr, tpr, thresholds = metrics.roc_curve(y_test, y_preds, pos_label=1)

metrics.auc(fpr, tpr)
```

Notice the `max_iter=6000` argument is not consistent with the grid search parameters. I increase it significantly in response to the model requiring more iterations to converge. Ultimately, the AUC of this model is `0.606`.

Conclusion

Considering the linear regression model, I applied was simply to provide an example of a regression setting and the results were extremely poor, I'll exclude it from model performance comparison. Therefore, I will compare only the four classification models (logistic regression, decision tree classifier, random forest classifier, and SVM). To conveniently arrange the different test AUC results in one dataframe I used the following user defined function.

```
def compare_models(models,preds,model_names):
    """This function makes evaluates predictions for a list of models,
    scores the models using AUC, and returns a dataframe with the class
    an AUC for each model and its predictions.
    Args: A list of model variables, a list of corresponding prediction values,
and a list of model names in string form.
    """
    scoring_df = pd.DataFrame()
    for ind, mod in enumerate(models):
        fpr, tpr, thresholds = metrics.roc_curve(y_test, preds[ind], pos_label=1)
        model_auc = metrics.auc(fpr, tpr)
        scoring_df[model_names[ind]] = [round(model_auc,2)]
    scoring_df.index = ['Test AUC']
    return(scoring_df)
```

The dataframe returned by this function can be seen below:

	Logistic Regression	Decision Tree Classifier	Random Forest Classifier	SVM
Test AUC	0.61	0.64	0.66	0.61

As can be seen above the random forest performed the best in this benchmarking application followed closely by the decision tree classifier, and finally logistic regression and SVM tie each other for the least performant AUC across these four classification algorithms. This comes as no surprise as tree-based methods are known for the consistently solid results.

Bibliography

Advantages and disadvantages of logistic regression. GeeksforGeeks. (2020, September 2). Retrieved March 30, 2022, from https://www.geeksforgeeks.org/advantages-and-disadvantages-of-logistic-regression/

Forward feature selection: Implementation of Forward Feature Selection. Analytics Vidhya. (2021, April 9). Retrieved March 30, 2022, from https://www.analyticsvidhya.com/blog/2021/04/forward-feature-selection-and-its-implementation/

Yiu, T. (2021, September 29). Understanding random forest. Medium. Retrieved March 30, 2022, from https://towardsdatascience.com/understanding-random-forest-58381e0602d2

Brownlee, J. (2021, April 26). One-vs-rest and one-vs-one for multi-class classification. Machine Learning Mastery. Retrieved March 30, 2022, from https://machinelearningmastery.com/one-vs-rest-and-one-vs-one-for-multi-class-classification/

Learn. scikit. (n.d.). Retrieved March 30, 2022, from https://scikit-learn.org/stable/