DSCI 503 – HW 08 Instructions

General Instructions

Create a new notebook named **`HW_08_YourLastName.ipynb`**. Download the files **`diamonds.txt`** and **`census.txt`** into the same directory as this notebook. Complete Parts 1 and 2 described below.

Any set of instructions you see in this document with an orange bar to the left will indicate a place where you should create a markdown cell. For each new problem, create a markdown cell that indicates the title of that problem as a level 2 header.

Any set of instructions you see with a blue bar to the left will provide instructions for creating a **single** code cell.

Read the instructions for each problem carefully. Each problem is worth 6 points. An additional 2 points are allocated for formatting and following general instructions.

Any time that you are asked to display a DataFrame, you should do so without using the **`print()`** function.

Assignment Header

Create a markdown cell with a level 1 header that reads: "DSCI 503 - Homework 08". Add your name below that as a level 3 header

Import the following packages using the standard aliases: **`numpy`**, **`pandas`**, and **`matplotlib.pyplot`**. Import the following classes and functions from the relevant modules in Scikit-Learn: **`train_test_split`**, **`OneHotEncoder`**, **`LinearRegression`**, **`LogisticRegression`**, **`DecisionTreeClassifier`**, **`RandomForestClassifier`**, **`classification_report`**, and **`confusion_matrix`**. No other packages should be used in this assignment.

In this assignment, you will be using scikit-learn to create and evaluate regression and classification models.

Problem 1: Diamonds Dataset

In Problem 1, you will be working with the Diamonds Dataset. This dataset contains information about several thousand diamonds sold in the United States. You can find more information about this dataset, including a description of its columns, here: Diamonds Dataset.

Load the data stored in the tab-delimited file **`diamonds.txt`** into a DataFrame named **`diamonds`**. As we have done in the past, add two new columns to **`diamonds`** named **`ln_carat`** and **`ln_price`**. These columns should contain the natural logarithms of the **`carat`** and **`price`** columns. Use **`head()`** to display the first 5 rows of this DataFrame.

Our goal is to create and compare two linear regression models to estimate the label **`ln_price`**. The first model will use **`ln_carat`** as the only feature. The second will use **`ln_carat`**, **`cut`**, **`color`**, and **`clarity`** as features. To use the categorical variables in a model, we will need to encode them using one-hot encoding.

Perform the following steps in a single code cell:

- Create a 2D array named **`X1_num`** by selecting the **`ln_carat`** column from **`diamonds`**.
- Create a 2D array named **`X1_cat`** by selecting the **`cut`**, **`color`**, and **`clarity`** columns from **`diamonds`**.
- Create a 1D array named **`y1`** by selecting the **`ln_price`** column from diamonds.
- Print the shapes of all three of these arrays with messages as shown below. Add spacing to ensure that the shape tuples are left-aligned.

```
Numerical Feature Array Shape:   xxxx
Categorical Feature Array Shape: xxxx
Label Array Shape:               xxxx
```

Note: The variables created here should be arrays, and not DataFrames or Series. You will need to use `.values`.

We will now perform one-hot encoding on the categorical variables.

Perform the following steps in a single code cell:

1. Create a **`OneHotEncoder()`** object setting **`sparse=False`**.
2. Fit the encoder to the categorical features.
3. Use the encoder to encode the categorical features, storing the result in a variable named **`X1_enc`**.
4. Print the shape of **`X1_enc`** with a message as shown below.

```
Encoded Feature Array Shape: xxxx
```

We will now combine the numerical feature array with the encoded categorical feature array.

Perform the following steps in a single code cell:

1. Use **`np.hstack`** to combine **`X1_num`** and **`X1_enc`** into a single array named **`X1`** with the numerical column appearing first in the new array.
2. Print the shape of **`X1`** with a message as shown below.

```
Feature Array Shape: xxxx
```

We will now split the data into training, validation, and test sets, using an 80/10/10 split.

Perform the following steps in a single code cell:

- Use **`train_test_split()`** to split the data into training and holdout sets using an 80/20 split. Name the resulting arrays **`X1_train`**, **`X1_hold`**, **`y1_train`**, and **`y1_hold`**. Set **`random_state=1`**.
- Use **`train_test_split()`** to split the holdout data into validation and test sets using a 50/50 split. Name the resulting arrays **`X1_valid`**, **`X1_test`**, **`y1_valid`**, and **`y1_test`**. Set **`random_state=1`**.
- Print the shapes of **`X1_train`**, **`X1_valid`**, and **`X1_test`** with messages as shown below. Add spacing to ensure that the shape tuples are left-aligned.

```
Training Features Shape:   xxxx
Validation Features Shape: xxxx
Test Features Shape:       xxxx
```

Add a markdown cell with a level 3 header that reads: "**Linear Regression Model with One Feature**".

Perform the following steps in a single code cell:

1. Create a linear regression model named **`dia_mod_1`**.
2. Fit the model to the training data, **using only the first (numerical) column of `X1_train`**.
3. Calculate the r-squared values for the training and validation set. Note that when using the **`score()`** method for this model, you will need to provide it with only the first column of the feature array.
4. Print the results with messages as shown below. Add spacing to ensure that the scores are left-aligned. Round the scores to 4 decimal places.

```
Training r-Squared:   xxxx
Validation r-Squared: xxxx
```

We will now see if we can improve our model's performance by adding in the three categorical features.

Add a markdown cell with a level 3 header that reads: "**Linear Regression Model with Several Features**".

Perform the following steps in a single code cell:

1. Create a linear regression model named **dia_mod_2**.
2. Fit the model to the training data using all features in **`X1_train`**.
3. Calculate the r-squared values for the training and validation set.
4. Print the results with messages as shown below. Add spacing to ensure that the scores are left-aligned. Round the scores to 4 decimal places.

```
Training r-Squared:   xxxx
Validation r-Squared: xxxx
```

You should have seen the r-Squared values increase by nearly 0.05 when we added in the categorical features. While the variable **`ln_carat`** alone explains a large proportion of the variance in the target variable **`ln_price`**, the three categorical features can be used to explain a bit ore of the variance.

Score the model **dia_mod_2** using the test set. Print the result with a message as shown below. Round the score to 4 decimal places.

```
Testing r-Squared: xxxx
```

Problem 2: Census Dataset

In Problem 2, you ou will be using census data from 1994 to attempt to predict whether or not a person has an annual salary greater than $50,000 based on other information provided in the census. You can find a description of the dataset here: Census Dataset

Load the data stored in the tab-delimited file **`census.txt`** into a DataFrame named **`census`**. Use **`head()`** to display the first 10 rows of this DataFrame.

We will now check to see how many rows and columns there are in the DataFrame.

Print the shape of the **`census`** DataFrame.

The last column is named **`salary`**. Each entry in this column is a string equal to either **`'<=50K'`** or **`'>50K'`**. Our goal is to create and compare several classification models for the purposes of predicting to which of these two classes an individual belongs based on the values of the other columns, which will be used as features in our models.

Before creating any models, we will check the distribution of values in our target variable.

Without creating any new DataFrame variables, select the **`salary`** column, and then call its **`value_counts()`** method. Display the result.

We will now prepare our data by encoding the categorical features and splitting into training, validation, and test sets.

Add a markdown cell with a level 3 header that reads: "**Prepare the Data**".

We will start by separating the categorial and numerical features into different arrays. Note that the following 8 features are categorical in nature: **`workclass`**, **`education`**, **`marital_status`**, **`occupation`**, **`relationship`**, **`race`**, **`sex`**, and **`native_country`**. The remaining 6 features are numerical.

Perform the following steps in a single code cell:

- Create a 2D array named **`X2_num`** by selecting the columns of **`census`** that represent numerical features.
- Create a 2D array named **`X2_cat`** by selecting the columns of **`census`** that represent categorical features.
- Create a 1D array named **`y2`** by selecting the **`salary`** column.
- Print the shapes of all three of these arrays with messages as shown below. Add spacing to ensure that the shape tuples are left-aligned.

```
Numerical Feature Array Shape:   xxxx
Categorical Feature Array Shape: xxxx
Label Array Shape:               xxxx
```

Note: The variables created here should be arrays, and not DataFrames or Series. You will need to use `.values`.

We will now perform one-hot encoding on the categorical variables.

Perform the following steps in a single code cell:

1. Create a **`OneHotEncoder()`** object setting **`sparse=False`**.
2. Fit the encoder to the categorical features.
3. Use the encoder to encode the categorical features, storing the result in a variable named **`X2_enc`**.
4. Print the shape of **`X2_enc`** with a message as shown below.

```
Encoded Feature Array Shape: xxxx
```

We will now combine the numerical feature array with the encoded categorical feature array.

Perform the following steps in a single code cell:

1. Use **`np.hstack`** to combine **`X2_num`** and **`X2_enc`** into a single array named **`X2`**.
2. Print the shape of **`X2`** with a message as shown below.

```
Feature Array Shape: xxxx
```

We will now split the data into training, validation, and test sets, using an 70/15/15 split.

Perform the following steps in a single code cell:

- Use **`train_test_split()`** to split the data into training and holdout sets using an 70/30 split. Name the resulting arrays **`X2_train`**, **`X2_hold`**, **`y2_train`**, and **`y2_hold`**. Set **`random_state=1`**. **Use stratified sampling**.
- Use **`train_test_split()`** to split the holdout data into validation and test sets using a 50/50 split. Name the resulting arrays **`X2_valid`**, **`X2_test`**, **`y2_valid`**, and **`y2_test`**. Set **`random_state=1`**. **Use stratified sampling**.
- Print the shapes of **`X2_train`**, **`X2_valid`**, and **`X2_test`** with messages as shown below. Add spacing to ensure that the shape tuples are left-aligned.

```
Training Features Shape:   xxxx
Validation Features Shape: xxxx
Test Features Shape:       xxxx
```

We will now create and evaluate a logistic regression model.

Add a markdown cell with a level 3 header that reads: "**Logistic Regression Model**".

Perform the following steps in a single code cell:

1. Create a logistic regression model named **`lr_mod`** setting **`solver='lbfgs',`** and **`max_iter=1000`**. Set **`penalty='none'`**, unless that results in an error, in which case, set **`C=10e1000`**.
2. Fit your model to the training data.
3. Calculate the training and validation accuracy with messages as shown below. Add spacing to ensure that the accuracy scores are left-aligned. Round the scores to 4 decimal places.

```
Training Accuracy:   xxxx
Validation Accuracy: xxxx
```

We will now create and evaluate several decision tree models. We will use the validation score for these models to perform hyperparameter tuning.

Add a markdown cell with a level 3 header that reads: "**Decision Tree Models**".

Perform the following steps in a single code cell:

1. Create empty lists named **`dt_train_acc`** and **`dt_valid_acc`**. These lists will store the accuracy scores that we calculate for each model.
2. Create a range variable named **depth_range** to represent a sequence of integers from 2 to 30.
3. Loop over the values in **depth_range**. Every time the loop executes, perform the following steps.
    a. Use NumPy to set a random seed of 1. **This should be done inside the loop.**
    b. Create a decision tree model named **`temp_tree`** with **`max_depth`** equal to the current value from **`depth_range`** that is being considered.
    c. Fit the model to the training data.
    d. Calculate the training and validation accuracy for **`temp_tree`**, appending the resulting values to the appropriate lists.
4. Use **`np.argmax`** to determine the index of the maximum value in **`dt_valid_acc`**. Store the result in **`dt_idx`**.
5. Use **`dt_idx`** and **`depth_range`** to find the optimal value for the **`max_depth`** hyperparameter. Store the result in **`dt_opt_depth`**.
6. Use **`dt_idx`** with the lists **`dt_train_acc`** and **`dt_valid_acc`** to determine the training and validation accuracies for the optimal model found.
7. Display the values found in Steps 5 and 6 with messages as shown below. Add spacing to ensure that the values replacing the xxxx symbols are left-aligned. Round the accuracy scores to 4 decimal places.

```
Optimal value for max_depth:           xxxx
Training Accuracy for Optimal Model:   xxxx
Validation Accuracy for Optimal Model: xxxx
```

We will now plot the validation and training curves as a function of the **`max_depth`** parameter.

Create a figure with two line plots on the same set of axes. One line plot should plot values of **`dt_train_acc`** against **`depth_range`** and the other should plot values of **`dt_valid_acc`** against **`depth_range`**. The x-axis should be labeled "**Max Depth**" and the y-axis should be labeled "**Accuracy**". The plot should contain a legend with two items that read "**Training**" and "**Validation**".

We will now create and evaluate several random forest models. We will use the validation score for these models to perform hyperparameter tuning.

Add a markdown cell with a level 3 header that reads: "**Random Forest Models**".

Perform the following steps in a single code cell:

1. Create empty lists named **`rf_train_acc`** and **`rf_valid_acc`**. These lists will store the accuracy scores that we calculate for each model.
2. Loop over the values in **depth_range**. Every time the loop executes, perform the following steps.
    a. Use NumPy to set a random seed of 1. **This should be done inside the loop.**
    b. Create a random forest model named **`temp_forest`** with **`max_depth`** equal to the current value from **`depth_range`** that is being considered. Set the parameter **`n_estimators`** to 100.
    c. Fit the model to the training data.
    d. Calculate the training and validation accuracy for **`temp_forest`**, appending the resulting values to the appropriate lists.
3. Use **`np.argmax`** to determine the index of the maximum value in **`rf_valid_acc`**. Store the result in **`rf_idx`**.
4. Use **`rf_idx`** and **`depth_range`** to find the optimal value for the **`max_depth`** hyperparameter. Store the result in **`rf_opt_depth`**.
5. Use **`rf_idx`** with the lists **`rf_train_acc`** and **`rf_valid_acc`** to determine the training and validation accuracies for the optimal model found.
6. Display the values found in Steps 4 and 5 with messages as shown below. Add spacing to ensure that the values replacing the xxxx symbols are left-aligned. Round the accuracy scores to 4 decimal places.

```
Optimal value for max_depth:           xxxx
Training Accuracy for Optimal Model:   xxxx
Validation Accuracy for Optimal Model: xxxx
```

We will now plot the validation and training curves as a function of the **`max_depth`** parameter.

Create a figure with two line plots on the same set of axes. One line plot should plot values of **`rf_train_acc`** against **`depth_range`** and the other should plot values of **`rf_valid_acc`** against **`depth_range`**. The x-axis should be labeled "**Max Depth**" and the y-axis should be labeled "**Accuracy**". The plot should contain a legend with two items that read "**Training**" and "**Validation**".

We will now create our final model and will evaluate it on the test set.

Add a markdown cell with a level 3 header that reads: "**Evaluate Final Model**".

Of the three types of models considered, and the various hyperparameter values for those models, select as your final model the one that produced the highest score on the validation set.

Perform the following steps in a single code cell:

1. If your final model is a decision tree or random forest model, use NumPy to set a random seed of 1.
2. Recreate the best model you found, using the parameter values that produced that model. Store the resulting model in a variable named **`final_model`**.
3. Fit this model to the training set.
4. Print the training accuracy, validation accuracy, and test accuracy for the final model with messages as shown below. Add spacing to ensure that the accuracy scores are left-aligned, and round the accuracy scores to four decimal places.

```
Training Accuracy for Final Model:   xxxx
Validation Accuracy for Final Model: xxxx
Testing Accuracy for Final Model:    xxxx
```

We will get a more detailed look at our final model's performance on the test set by creating a confusion matrix and a classification report.

Use your final model to generate predictions for the test set, storing them in a variable named **`test_pred`**. Create a confusion matrix by passing **`y2_test`** and **`test_pred`** to the function **`confusion_matrix()`** from Scikit-Learn. This function returns a NumPy array. Store this array in a variable, and then convert it to a DataFrame with row and column names being set equal to the valid labels, **`'<=50K'`** and **`'>50K'`**. Display the resulting DataFrame.

Pass the arrays **`y2_test`** and **`test_pred`** to the function **`classification_report()`**, printing the result.

Submission Instructions

When you are done, click **Kernel > Restart and Run All**. If any cell produces an error, then manually run every cell after that one, in order. Save your notebook, and then export the notebook as an HTML file. Upload the HTML file to Canvas and upload the IPYNB file to CoCalc, placing it in the **Homework/HW 08** folder.