DSCI 503 – Project 04 Instructions

Background

In this project, we will be working with the Forest Cover dataset. Additional information about this dataset can be found here: Kaggle: Forest Cover Type Prediction

Your goal will be to create a model to generate predictions about the type of forest cover in a particular wilderness region based on cartographic information. The data you will be working with was collected from the Roosevelt National Forest in Northern Colorado. Each observation represents a 30 meter by 30 meter patch of land. The dataset contains the following columns:

- **`Elevation`** – The average elevation of the region, in meters.
- **`Aspect`** – A measure of the direction the slope of the region faces, in degrees, with 0 degrees indicating North.
- **`Slope`** – The average slope of the region, in degrees.
- **`Horizontal_Distance_To_Hydrology`** – The horizontal distance to the nearest surface water.
- **`Vertical_Distance_To_Hydrology`** – The vertical distance to the nearest surface water.
- **`Horizontal_Distance_To_Roadways`** – The horizontal distance to the nearest roadway.
- **`Hillshade_9am`** – A measure of the amount of shade the region has at 9 am during the summer solstice.
- **`Hillshade_Noon`** – A measure of the amount of shade the region has at noon during the summer solstice.
- **`Hillshade_3pm`** – A measure of the amount of shade the region has at 3 pm during the summer solstice.
- **`Horizontal_Distance_To_Fire_Points`** – The horizontal distance to the nearest wildfire ignition points.
- **`Wilderness_Area`** – A categorical variable indicating which of four wilderness areas the region resides in. The names of the four areas are: Rawah, Neota, Comanche Peak, and Cache la Poudre
- **`Soil_Type`** – A categorical variable indicating which of 40 soil types is predominant in the region. The soil types are encoded as integers with values 1 – 40. A list of the soil types can be found on the Kaggle side linked above.
- **`Cover_Type`** – A categorial variable indicating which of 7 forest cover types is predominant in the region. The cover types are encoded as integers with values 1 – 7. The names of the 7 types are provided below:

    1. Spruce/Fir
    2. Lodgepole Pine
    3. Ponderosa Pine
    4. Cottonwood/Willow
    5. Aspen
    6. Douglas-Fir
    7. Krummholz

General Instructions

Create a new notebook named **`Project_04_YourLastName.ipynb`** and complete the instructions provided 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. If no instructions are provided regarding formatting, then the text should be unformatted.

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 carefully.

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

Assignment Header

Create a markdown cell with a level 1 header that reads: "DSCI 503 – Project 04". 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`**, **`LogisticRegression`**, **`DecisionTreeClassifier`**, **`RandomForestClassifier`**, **`classification_report`**, and **`confusion_matrix`**. No other packages should be used in this project.

Part 1: Loading the Dataset; Preliminary Analysis

In this section, we will load the data into a DataFrame, and will explore the structure of the data set.

Create a markdown cell that displays a level 2 header that reads: "**Part 1: Loading the Dataset; Preliminary Analysis**". Also add some text briefly describing the purpose of your code in this part.

The data is stored in the tab-delimited text file **`forest_cover.txt`**. Download this file into the directory that contains your notebook, and then load the data into a DataFrame named **`fc`**. Use the **`head()`** method to display the first 5 rows of this DataFrame.

Add a markdown cell explaining that we will determine the size of the dataset. State what the results tell you about the number of observations in the dataset.

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

We will now inspect the distribution of cover types in the datasets. Add a markdown cell to briefly explain this.

Without creating any new DataFrame variables, select the **`Cover_Type`** column from **`fc`**, then call its **`value_counts()`** method, followed by the **`sort_index()`** method. Display the result.

You should see that the seven cover types are equally represented in the dataset. This is an important observation. If we were to create a model that just guesses the cover type at random, we would expect that it would be correct 1/7 of the time, and thus have an accuracy of 0.1429. This will provide us with an important benchmark for evaluating our models.

Add a markdown cell explaining that you are about to create a list of seven colors to be used as a palette in plots that you will create later.

Create a list named **`palette`** containing seven named colors. The colors should be easily distinguishable from each other. You are welcome to select your own colors, or to use the following list:

```
['orchid', 'lightcoral', 'orange', 'gold', 'lightgreen', 'deepskyblue', 'cornflowerblue']
```

Part 2: Distribution of Cover Type by Wilderness Area

In this section, we will explore the relationship between cover type and wilderness area.

Create a markdown cell that displays a level 2 header that reads: "**Part 2: Distribution of Cover Type by Wilderness Area**". Also add some text briefly describing the purpose of your code in this part. Explain that we will start by determining the distribution of the wilderness areas within our dataset.

Without creating any new DataFrame variables, select the **`Wilderness_Area`** column from **`fc`**, then call its **`value_counts()`** method, followed by the **`sort_index()`** method. Display the result.

Add a markdown cell explaining that we will create a DataFrame to determine the how many regions of each cover type are in each of the four wilderness areas.

Use **`pd.crosstab()`** to count the number of regions of each cover type that are in each of the four wilderness areas. Pass this function the **`Cover_Type`** column as its first argument and the **`Wilderness_Area`** column as the second argument. Store the results in a DataFrame named **`ct_by_wa`** and then display this DataFrame.

Add a markdown cell explaining that you will visually represent the information in the DataFrame you just created in the form of a stacked bar chart.

Perform the following steps in a single cell:

1. Start by converting the count information into proportions. Create a DataFrame named **`ct_by_wa_props`** by dividing **`ct_by_wa`** by the column sums of **`ct_by_wa`**. The column sums can be calculated using **`np.sum()`** or the DataFrame **`sum()`** method.
2. We will be creating a stacked bar chart, so we need to know where the bottom of each bar should be located. This can be calculated as follow: **`bb = np.cumsum(ct_by_wa_props) - ct_by_wa_props`**
3. Create a Matplotlib figure, setting the figure size to `[8, 4]`.
4. Loop over the rows of **`ct_by_wa_props`**. Each time this loop executes, add a bar chart to the figure according to the following specifications.
   - The height of the bars should be determined by the current row of **`ct_by_wa_props`**.
   - The bottom position of each bar should be determined by the current row of **bb**.
   - Each bar should have a black border, and a fill color determined by the current value of **`palette`**.
   - The label for the legend should be set to the value of **`Cover_Type`** associated with the current row.
5. Set the labels for the x and y axes to be **"Wilderness Area"** and **"Proportion"**. Set the title to be **"Distribution of Cover Type by Wilderness Area"**.
6. Add a legend to the plot. Set the **`bbox_to_anchor`** parameter to place the legend to the right of the plot, near the top.
7. Display the figure using **`plt.show()`**.

Note: You performed a task very similar to this in Problem 8 of Homework 05.

Part 3: Distribution of Cover Type by Soil Type

In this section, we will explore the relationship between cover type and soil type.

Create a markdown cell that displays a level 2 header that reads: "**Part 3: Distribution of Cover Type by Soil Type**". Also add some text briefly describing the purpose of your code in this part. Explain that we will start by creating a DataFrame to determine the number of regions of each cover type there are for each of the 40 soil types.

Use **`pd.crosstab()`** to count the number of regions of each cover type there are for each of the 40 soil types. Pass this function the **`Cover_Type`** column as its first argument and the **`Soil_Type`** column as the second argument. Store the results in a DataFrame named **`ct_by_st`** and then display this DataFrame.

Add a markdown cell explaining that you will visually represent the information in the DataFrame you just created in the form of a stacked bar chart.

Repeat the steps from the last cell in Part 2, with the following changes:

- Use **`ct_by_st`** instead of **`ct_by_wa`**.
- Name the newly created DataFrame **`ct_by_st_props`** instead of **`ct_by_wa_props`**.
- Use **`ct_by_st_props`** instead of **`ct_by_wa_props`**.
- Set the figure size to be `[12,6]`.
- Change the words "**Wilderness Area**" to "**Soil Type**" in the x-axis label and in the title.

Part 4: Distribution of Elevation by Cover Type

In this section, we will explore the relationship between cover type and elevation.

Create a markdown cell that displays a level 2 header that reads: "**Part 4: Distribution of Elevation by Cover Type**". Also add some text briefly describing the purpose of your code in this part. Explain that we will start by calculating the average elevation for each of the seven cover types.

Select the **`Elevation`** and **`Cover_Type`** columns from **`fc`**, group the rows by **`Cover_Type`**, and then calculate the mean **`Elevation`** for each group. Display the resulting DataFrame.

Add a markdown cell explaining that you will create histograms to visually explore the distribution of elevations for each of the seven cover types.

Create a figure containing seven subplots arranged in a 2x4 grid (the bottom right subplot will be empty). Each of the subplots should contain a histogram of elevations for regions with one specific cover type. This can be accomplished with the following steps:

1. Set a figure size of `[12,6]`.
2. Loop over the possible values of **`Cover_Type`** (which are integers, 1 – 7). Each time the loop executes:
   a. Create a new subplot.
   b. Add a histogram of **`Elevation`**, using only the observations corresponding to the current value of **`Cover_Type`**. Set **`bins=np.arange(1800, 4050, 50)`** and set the fill color for the bars to be equal to the corresponding color in **`palette`**. You can add a black border or not, depending on what you think looks better.
   c. Set the title of the subplot to be "**Cover Type X**", with "**X**" replaced by the appropriate integer.
   d. Set the x-limits to `[1800,4000]` and set the y-limits to be `[0,600]`.
   e. Set the axis labels for each subplot to "**Elevation**" and "**Count**".
3. After the loop finishes, call **`plt.tight_layout()`**, and then use **`plt.show()`** to display the figure.

Part 5: Creating Training, Validation, and Test Sets

In this section, we will encode our categorical variables and will create training, validation, and test sets.

Create a markdown cell that displays a level 2 header that reads: "**Part 5: Creating Training, Validation, and Test Sets**". Also add some text briefly describing the purpose of your code in this part. Explain that we will start by separating the categorical features, the numerical features, and the labels.

Before moving on to the next step, note that we will be using **Cover_Type** as the label variable in our models. All other columns will be used as features. Of the feature columns, **Wilderness_Area** and **Soil_Type** are categorical, while all other feature columns are numerical.

Perform the following steps in a single code cell:

- Create a 2D array named **`X_num`** by selecting the columns of **`fc`** that represent numerical features.
- Create a 2D array named **`X_cat`** by selecting the columns of **`fc`** that represent categorical features.
- Create a 1D array named **`y`** by selecting the column of **`fc`** corresponding to the labels.

- 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`.

Create a markdown cell explaining that we will now be encoding the categorical variables using one-hot encoding.

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 **`X_enc`**.
4. Print the shape of **`X_enc`** with a message as shown below.

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

Create a markdown cell explaining that we will now combine the numerical features with the encoded features.

Perform the following steps in a single code cell:

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

```
Feature Array Shape: xxxx
```

Create a markdown cell explaining that we will now split the data into training, validation, and test sets, using a 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 **`X_train`**, **`X_hold`**, **`y_train`**, and **`y_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 **`X_valid`**, **`X_test`**, **`y_valid`**, and **`y_test`**. Set **`random_state=1`**. **Use stratified sampling**.
- Print the shapes of **`X_train`**, **`X_valid`**, and **`X_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
```

Part 6: Logistic Regression Model

In this section, we will create and evaluate a logistic regression model.

Create a markdown cell that displays a level 2 header that reads: "**Part 6: Logistic Regression Model**". Also add some text briefly describing the purpose of your code in this part.

Perform the following steps in a single code cell:

1. Create a logistic regression model named **`lr_mod`** setting **`solver='lbfgs',max_iter=1000`**, and **`multi_class='multinomial'`**. Set **`penalty='none'`**. Your model will likely not converge with the number of iterations we have selected, but for the sake of execution time, will will not worry about increasing it.
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
```

Part 7: Decision Tree Models

In this section, we will create and evaluate several decision tree models.

Create a markdown cell that displays a level 2 header that reads: "**Part 7: Decision Tree Models**". Also add some text briefly describing the purpose of your code in this part.

We will now create several decision tree models, each with a different value for the **`max_depth`** parameter. For each parameter value, we will calculate the training and validation accuracy. We will use the validation scores to select the optimal value for **`max_depth`**.

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
```

Create a markdown cell explaining that you will plot the training and validation curves as a function of **`max_depth`**.

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**".

Part 8: Random Forest Models

In this section, we will create and evaluate several random forest models.

Create a markdown cell that displays a level 2 header that reads: "**Part 8: Random Forest Models**". Also add some text briefly describing the purpose of your code in this part.

We will now create several random forest models, each with a different value for the **`max_depth`** parameter. For each parameter value, we will calculate the training and validation accuracy. We will use the validation scores to select the optimal value for **`max_depth`**.

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** (from Part 7). 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_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
```

Create a markdown cell explaining that you will plot the training and validation curves as a function of **`max_depth`**.

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**".

Part 9: Create and Evaluate Final Model

In this section, we will select our final model, and will evaluate it on the test set.

Create a markdown cell that displays a level 2 header that reads: "**Part 8: Random Forest Models**".

Add some text explaining which of the three types of models (logistic regression, decision tree, or random forest) you will use as your final model, and state what parameter values you will you. Select the model that gave you the best performance 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
```

Create a markdown cell explaining that we will now create and display a confusion matrix detailing the model's performance on the test set.

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 **`y_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 (i.e. the integers 1 – 7). Display the resulting DataFrame.

Create a markdown cell explaining that we will now generate a classification report to provide further insight into the model's performance on the test set.

Pass the arrays **`y_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 **Projects/Project 04** folder.