COM SCI M148 Final Project Report: Evergreen Project Group

Rama Das, Chloe DePaul, Dagny Parayao, and Dane Sheridan

December 19, 2024

1 Introduction

Our project focused on developing an optimal predictive model to identify forest cover types and determine the key environmental factors that drive their distribution. We went through a series of comprehensive data modeling techniques with the following documentation of methodologies and findings.

The dataset includes four wilderness areas located in the Roosevelt National Forest of northern Colorado and the forest cover type for each 30 x 30 meter area was identified using data from the US Forest Service (USFS) Region 2 Resource Information System (RIS). The predictor variables were derived from US Geological Survey (USGS) and USFS datasets. This dataset was made accessible through the UCI Machine Learning Repository.

```
Number of records of Spruce-Fir:          211840
Number of records of Lodgepole Pine:      283301
Number of records of Ponderosa Pine:      35754
Number of records of Cottonwood/Willow:   2747
Number of records of Aspen:               9493
Number of records of Douglas-fir:         17367
Number of records of Krummholz:           20510
Number of records of other:               0
Total records:                            581012
```

Figure 1: Distribution of Forest Cover Types

2 Methodology and Results

Performing principal component analysis (PCA) was highly effective in determining which environmental features best predict forest cover type. In particular, the proportion of cumulative variance in the original data that is explained by the first two principal components (PCs) we computed is over 96 percent. In these first two principal components, the top two features with respect to variable loadings were horizontal distance to roadways and horizontal distance to fire points (A.6), with elevation a distant third. This analysis motivates our use of these features, as well as several other significant features identified in the PCA transformation, as predictors in our logistic regression analysis (A.4).

K-Nearest Neighbors (KNN) was the most effective approach to helping us address our other goal of predicting forest cover types. As detailed in A.5, due to the inherent nature of forest cover type distribution, we were able to achieve a test accuracy of 89.77% with the KNN model. In comparison, our neural network model was only able to achieve 77% accuracy with the training and validation sets, as seen in A.7. And A.4 shows that our logistic regression model was only able to achieve an accuracy of 67.67%, even with optimized feature selection and applying ridge regression.

When building our various models, we employed multiple cross-validation and evaluation techniques to ensure our models were robust. For KNN, we used different k values (1, 5, 15, and 25). From this, we found that k=5 provided the optimal balance between training and test accuracy. (A.5) Furthermore, we also evaluated the model's performance with confusion matrices, which showed us that certain covertypes had better prediction accuracy than others. For our neural network (A.7), we validated results by running a training set and validation set. We also chose a mini batch approach, which allows us to sample different subsets of the data during training and experimented with different hyperparameters as well as batch sizes. We also employed cross-validation for ridge regression in order to determine the optimal alpha value. (A.3)

3 Using Our Code

All of our code can be found in a public github repository: https://github.com/ramafications/Evergreen. The code can be run simply through cloning the repository and navigating to the main branch which has a single jupyter notebook. We have centralized all the following data models beginning from initial data analysis to the final neural network model. Hitting run all in the notebook will generate all working data models and methods of evaluation/performance we have used.

Appendices

A.1 Exploratory Data Analysis

Our exploratory data analysis (EDA) began with reading the dataset information provided to gain a better understanding of the data that we were working with. Based on this reading, we learned that the forest areas researched in this study were four wilderness areas located in the Roosevelt National Forest of northern Colorado. These were areas with minimal human interaction so we can assume that the cover types in these areas were influenced by the natural environmental factors rather than man-made. Our project aims to discover how we can best predict these cover types based on the set of environmental factors collected in this data set. There were seven different types of trees, or cover types, recorded in these data, which we list here:

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

Figure 2: Forest Cover Type Classifications

Once we had the background information we loaded the data set and began our initial EDA process. We first displayed the dimensions of the data set and printed the column names to gain an understanding of how the data was organized and to make sure that we understood what each feature was representing. We verified that the data was clean, as we know this preprocessing step was done for us, to make sure that there were no large gaps anywhere in our data.

Name	Measurement	Description
Elevation	meters	Elevation in meters
Aspect	azimuth	Aspect in degrees azimuth
Slope	degrees	Slope in degrees
Horizontal Distance To Hydrology	meters	Horz Dist to nearest surface water features
Vertical Distance To Hydrology	meters	Vert Dist to nearest surface water features
Horizontal Distance To Roadways	meters	Horz Dist to nearest roadway
Hillshade 9am	0 to 255 index	Hillshade index at 9am, summer solstice
Hillshade Noon	0 to 255 index	Hillshade index at noon, summer soltice
Hillshade 3pm	0 to 255 index	Hillshade index at 3pm, summer solstice
Horizontal Distance To Fire Points	meters	Horz Dist to nearest wildfire ignition points
Wilderness Area (4 binary columns)	0 (absence) or 1 (presence)	Wilderness area designation
Soil Type (40 binary columns)	0 (absence) or 1 (presence)	Soil Type designation
Cover Type	Classes 1 to 7	Forest Cover Type designation - *Response Variable*

Figure 3: Feature Information

We then began the visualizations of our data. We began with a stacked histogram of Tree/Cover type against each of the non categorical features to see how the tree types were distributed amongst each feature. We then made this into a density curve for easier visualization. This was used to give us insights into how each feature behaves across the 7 different tree types. Features were visualized individually using a subplot grid, with each subplot displaying the distribution of a specific feature for each of the tree types.

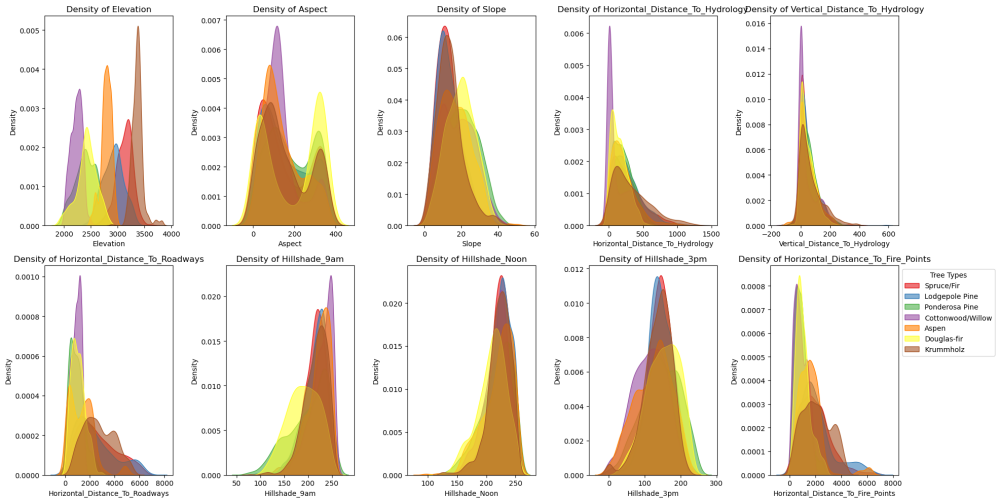


Figure 4: Density Curves of Select Features

We then used a bar plot to visualize the distribution of the target variable `covertype`. This allowed us to better understand the frequency of each tree type in the data set which would help us to make sure that we are taking representative samples later on.

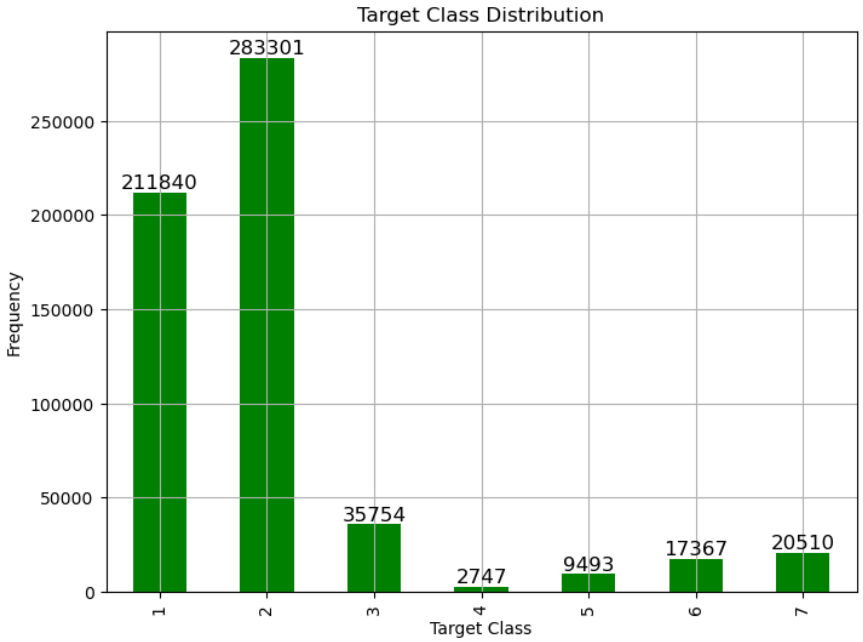


Figure 5: Cover Type Distribution

We also created a bar plot to visualize the different soil types. Since this was a binary variable this plot allowed us to see the frequency at which each soil type occurred. This allowed us to ignore soil types that were near 0 in order to declutter our models later on.

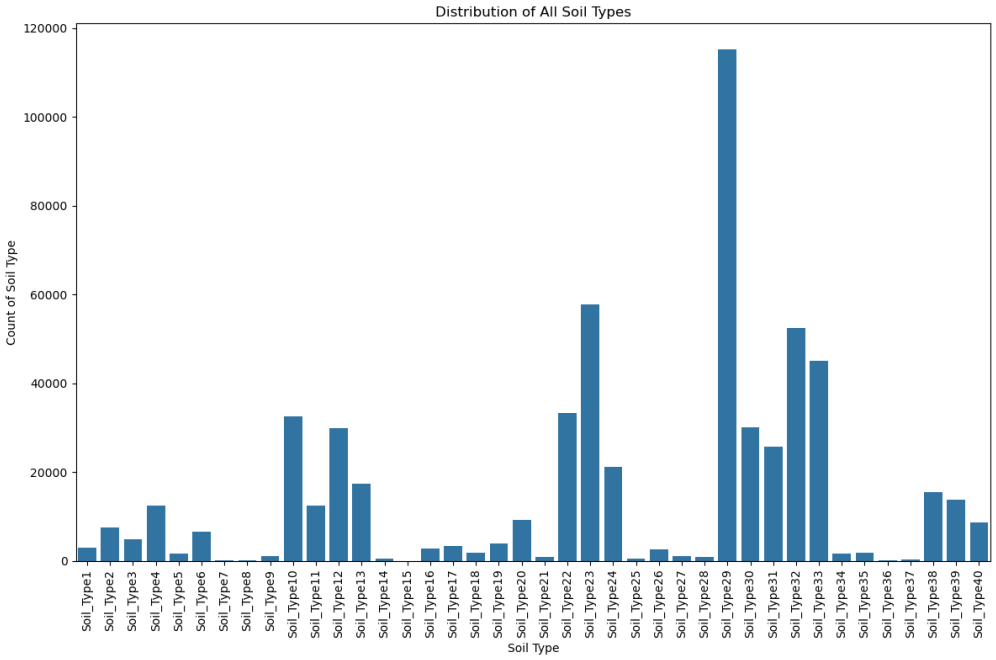


Figure 6: Soil Distribution

A.2 Data Preprocessing and Feature Engineering

The dataset was split into two parts: X (features): This includes all the columns except for the target variable `Target`, which contains the forest cover type. y (target): The target variable that represents the forest cover type (from 1 to 7). The features X and the target y were combined into a single DataFrame `df`, which facilitated easier exploration and visualization of the data. We then split the data into train, test and validation sets in order to better train the model.

A.3 Regression Analysis

LAD Regression was applied on a subset of data to model the relationship between different independent variables (features) and the dependent variable `Target`. From each regression model, we could observe the strength of the linear relationships between the independent variable and the target variable. We learned that some features, like `Elevation`, had a noticeable linear relationship with the target, while others might have a weaker or less clear association. The intercept and coefficient values gave us insight into how much influence each feature had on the target variable.

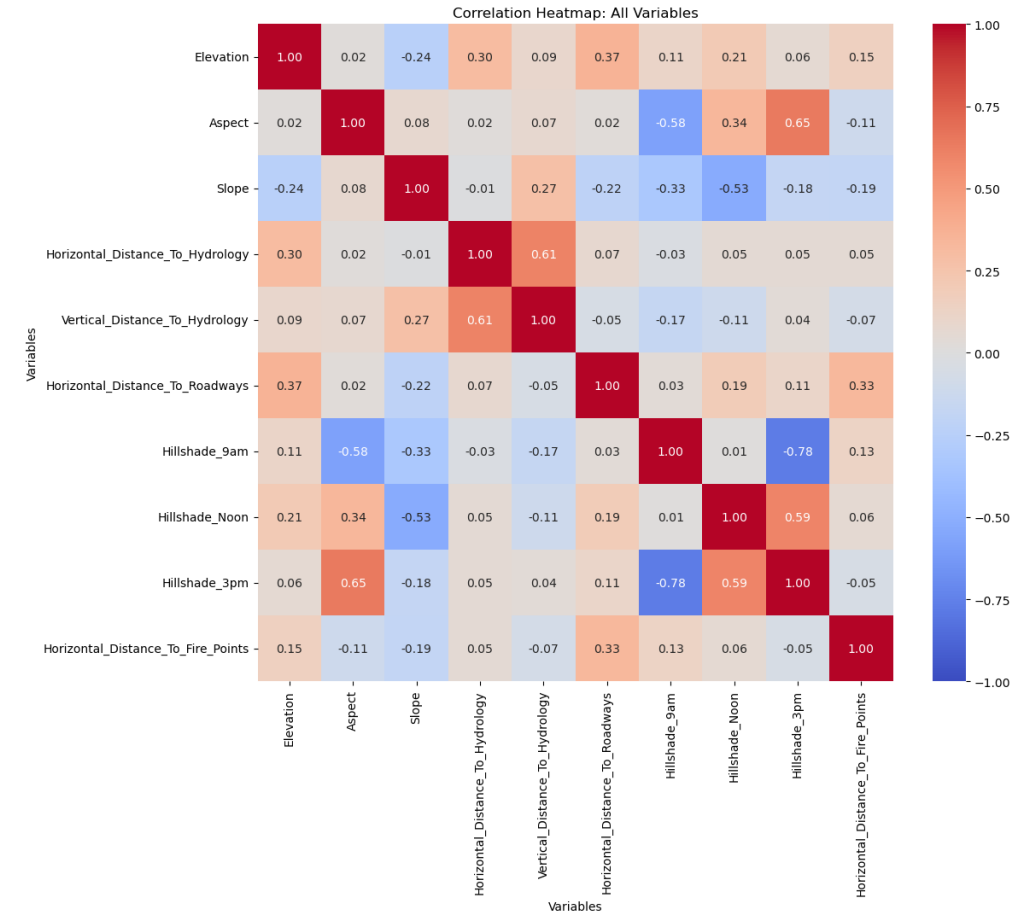
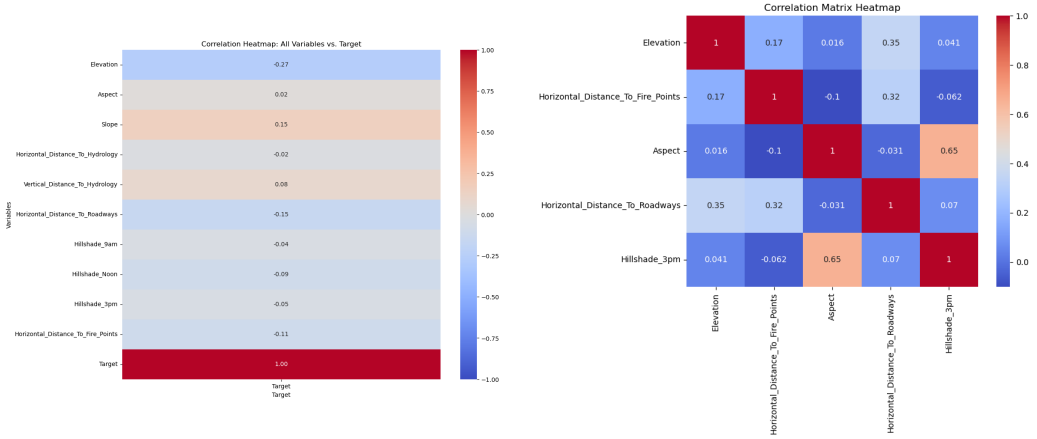


Figure 7: Correlation Heatmap

A correlation matrix was computed for the dataset, excluding binary variables (such as those representing soil types and wilderness areas), and the matrix was visualized using a heatmap. We also computed the correlation specifically between the target variable `Target` and the other features to identify which features were more strongly correlated with the target. Additionally, we evaluated the collinearity matrix to make sure that none of these predictor variables had a strong correlation with one another that may be introducing redundant information.



Ridge Regression was used to handle multicollinearity and prevent overfitting. The dataset was standardized using `StandardScaler` to ensure all features were on the same scale. This is essential for regularized regression models like Ridge, as the penalty term depends on the scale of the features. A range of alpha values was tested for Ridge regression to find the optimal regularization strength. This was done using cross-validation, where the negative root mean squared error (rMSE) was calculated for each alpha. The optimal alpha value that minimized the test rMSE was chosen, helping to balance the model's fit and its ability to generalize to unseen data. Ridge regression helps reduce the variance in the model which helps to prevent overfitting. This was particularly useful when dealing with multicollinearity and ensuring that the model could generalize well to new data. Based on the correlation matrix, non of the features were highley correlated, however some were approaching a high correlation. Regularization helps to control their combined influence. Ridge regression provided a more stable solution by shrinking the coefficients of less important features.

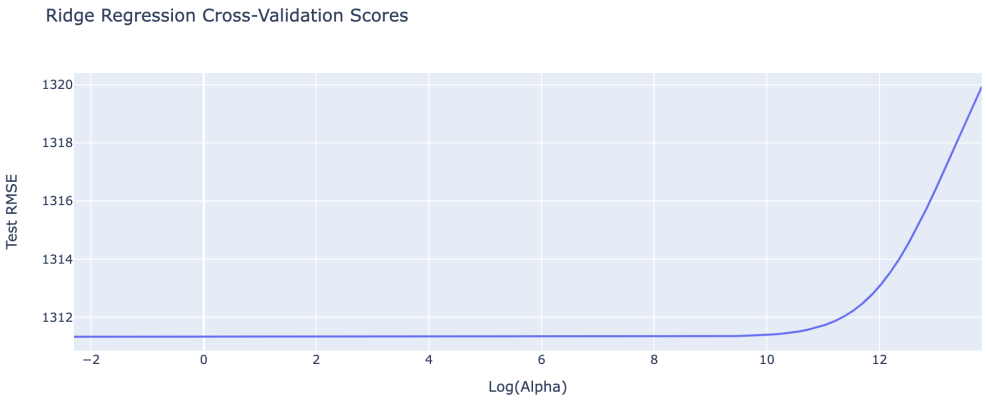


Figure 8: Ridge Regression Plot

From the cross-validation scores, we learned how the alpha value affects the model's performance. A low alpha leads to less regularization, while a high alpha introduces more regularization which would shrink the coefficients of variables more rapidly. The best alpha is the one that gives the lowest rMSE on the validation set.

A.4 Logistic Regression Analysis

We performed logistic regression with multiclass classification to predict forest cover type from a subset of the remaining features in the dataset. For our predictors, we first used the top five features in the first component of the PCA-transformed dataset (Fig. 13). Training a logistic regression model on our training set with these five predictors yielded an accuracy score of approximately 0.6437. Adding several more variables yielded a score of 0.6767, the highest accuracy metric we were able to obtain from any combination of predictor features.

In Fig. 9, we plot the receiving operator characteristic (ROC) curves for the seven cover types, obtained by binarizing (i.e. one-hot encoding) the target variable and comparing the resulting values to the logistic regression model's predicted probabilities for each class. Additionally, we display the expected performance of a random classifier, i.e. a classifier equally likely to output a true positive or false positive, for comparison.

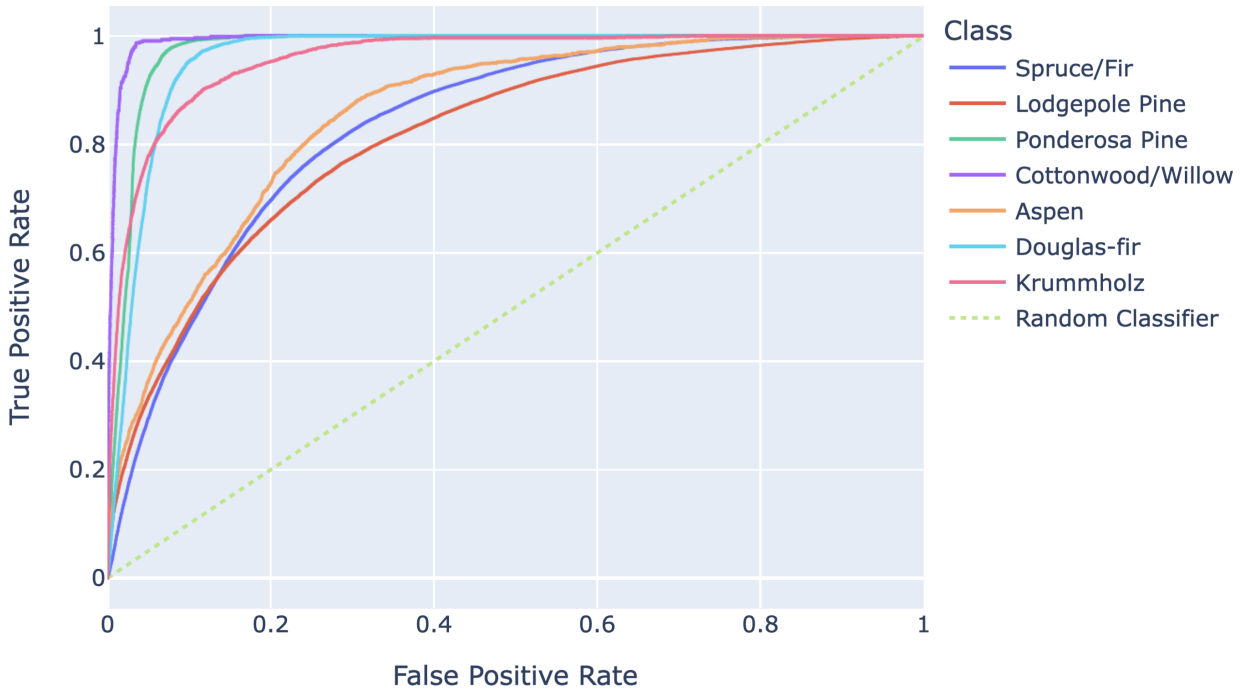


Figure 9: ROC Curves for Each Forest Cover Type

The ROC performance metrics for these data are very good, even better than the accuracy metric given above. The respective areas under each curve (AUC), as well as the average AUC over all seven curves, are given in Fig. 10. Since the cover type distribution is dominated by the Spruce/Fir and Lodgepole Pine classes (see Fig. 11), the discrepancy between our model's accuracy metric and its averaged AUC results is not surprising: In general, ROC is less affected by skewness in the distribution of the target variable than a simple proportion of class prediction accuracy, and it is consequently a better indicator of performance for unbalanced datasets such as ours.

```
Spruce/Fir: 0.8340
Lodgepole Pine: 0.8141
Ponderosa Pine: 0.9760
Cottonwood/Willow: 0.9928
Aspen: 0.8554
Douglas-fir: 0.9620
Krummholz: 0.9580
Average AUC: 0.9132
```

Figure 10: AUC for Each Forest Cover Type

A.5 Classification

The nature of our dataset and importance of neighboring data points influencing cover type classification, made it a natural candidate for k-Nearest Neighbors. Determining the optimal number of variables for the KNN model was a crucial decision, and incorporating PCA-transformed variables proved to be highly beneficial for model performance. The first attempt to run all the variables was

extremely time inefficient, due to euclidean distance algorithm running for each data point against its k neighbors. The initial run of the two largest variables of variance explained by the PCA: distance to roadways and distance to fire points, led to a 65.2 accuracy, however when including a set of other variables accuracy shot up to 89.76 percent. The final variables selected to train the model were: elevation, aspect, slope, horizontal and vertical distances to water bodies, distance to roadways, hillshade measurements taken at 9am, noon, and 3pm, distance to fire points, and four different wilderness area designations.

After deciding the set of predictors it was vital to select the most optimal k and test a variety of k neighbors. An initial range of values of k = 1, k =5, k = 15, and k = 25 models were trained and evaluated on test and train accuracies. The following figure below represents the calculated classification accuracies on k models.

```
Classification accuracy for knn1 were:
 Train = 1.0 , Test = 0.907618829063997
Classification accuracy for knn5 were:
 Train = 0.9372680495544758 , Test = 0.8976793551533232
Classification accuracy for knn15 were:
 Train = 0.8939132617387096 , Test = 0.8740426264306819
Classification accuracy for knn25 were:
 Train = 0.8728077881564086 , Test = 0.8579214595106279
```

Figure 11: Classification Accuracy for KNN models

The figure below is an assembled confusion matrix of true positive, true negative, false positive, and false negative rate. The dipping accuracies of 4, 5, and 6 cover types are clearly aligned with the frequency of those cover types in the data set, thus this lack of data may have explained the model lower prediction accuracy(less neighbors of true cover type in vicinity). The evaluation of the model organized by cover type concluded that the KNN model did not perform uniformly across cover types and further optimization of the model may have been done simply by having more data of 4,5, and 6 cover types. Overall this was the better classification method outperforming decision trees. This was most likely due to the nature of the variables, as their relationships with forest cover type are likely to be gradual and continuous rather than sharp decision boundaries. It intuitively follows the idea that similar instances have similar cover types.

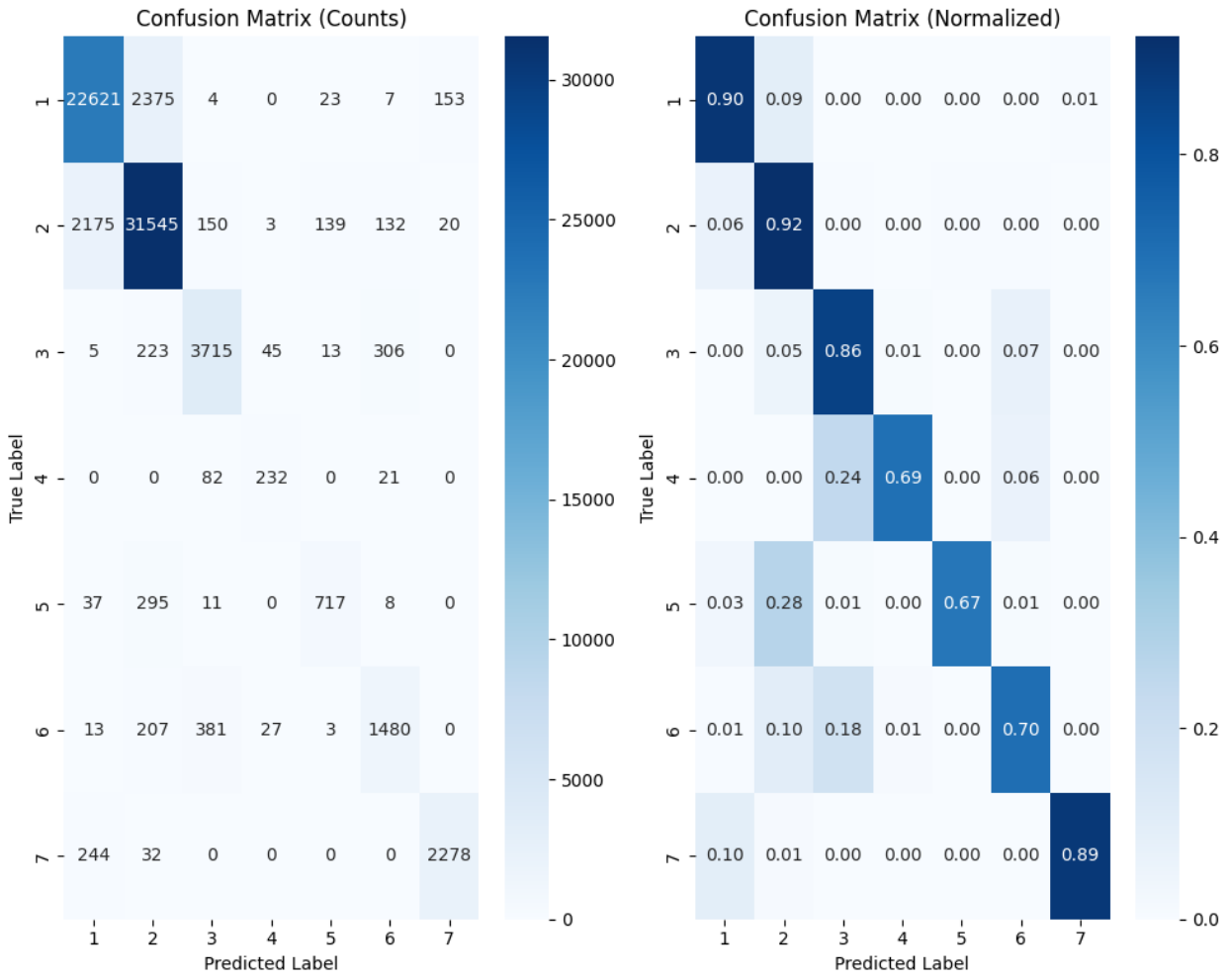


Figure 12: KNN Confusion Matrix per Cover Type

A.6 Principal Component Analysis

We used PCA as part of our classification pipeline to determine what features best predict forest covertype. Applying PCA helps us reduce dimensionality from the original 54 features by determining the most relevant features and how they are connected to our predictor variable. In our graphs from exploratory data analysis, we were only able to use visual cues to determine that the most relevant feature might be `Elevation`, `Distance to Roadways`, and `Firepoints` but PCA helps us confirm our hypotheses.

```
Cumulative variance explained by components:
First 1 components explain 66.25% of variance
First 2 components explain 96.98% of variance

Top 5 features in component 1:
Horizontal_Distance_To_Roadways: 0.8488
Horizontal_Distance_To_Fire_Points: 0.5253
Elevation: 0.0590
Horizontal_Distance_To_Hydrology: 0.0103
Vertical_Distance_To_Hydrology: -0.0022

Top 5 features in component 2:
Horizontal_Distance_To_Fire_Points: 0.8508
Horizontal_Distance_To_Roadways: -0.5245
Elevation: -0.0296
Aspect: -0.0122
Hillshade_3pm: -0.0041
```

Figure 13: Cumulative Explained Variance and Top Features for First Two PCs

As we can see in Fig. 13, horizontal distance to roadways and horizontal distance to fire points explained 96.98 percent of cumulative variance in the dataset. Running these two features as variables instead of all 54 features on our logistical regression and neural network models significantly improved the output and efficiency of these models, demonstrating the importance of PCA in our project.

On the other hand, clustering would not have been helpful for addressing our problem of accurately predicting forest covertype, as it's used as a method to group together data points. While clustering may have told us what factors group together data points, it's more efficient to use PCA to determine which features contribute most to this prediction rather than analyze clustering results.

We also did not run cross-validation on PCA as the goal of PCA is dimensionality reduction not prediction.

A.7 Neural Network

We implemented neural networks to classify observations in our dataset as being a certain `Target`. After experimenting with different implementations of the network, we found that dropping certain columns made the network perform much faster without losing accuracy. For this reason, we dropped the columns containing Soil Types and Wilderness Areas. Then we made sure to standardize the data to ensure that all input features are on the same scale in order to properly evaluate our results.

We implemented the network to be a multi-layer perceptron (MLP) with one hidden layer. To initialize the network we specified the number of features, the number of hidden layers (96) and the number of classes (7). The model then implements forward and backward propagation with a learning rate of 0.01.

To train the model we utilized mini-batch gradient descent. In this process the data is divided into mini-batches (random subsets of the training set) and the model parameters are updated iteratively based on the average across each mini-batch. This process runs for 50 epochs, during which the training and validation loss were computed to track the models performance. We then plotted the Training Accuracy and Loss as well as the Validation Accuracy and Loss to make sure that they were stable and performed similarly. If the accuracy or loss were to diverge suddenly this would indicate a problem with our model.

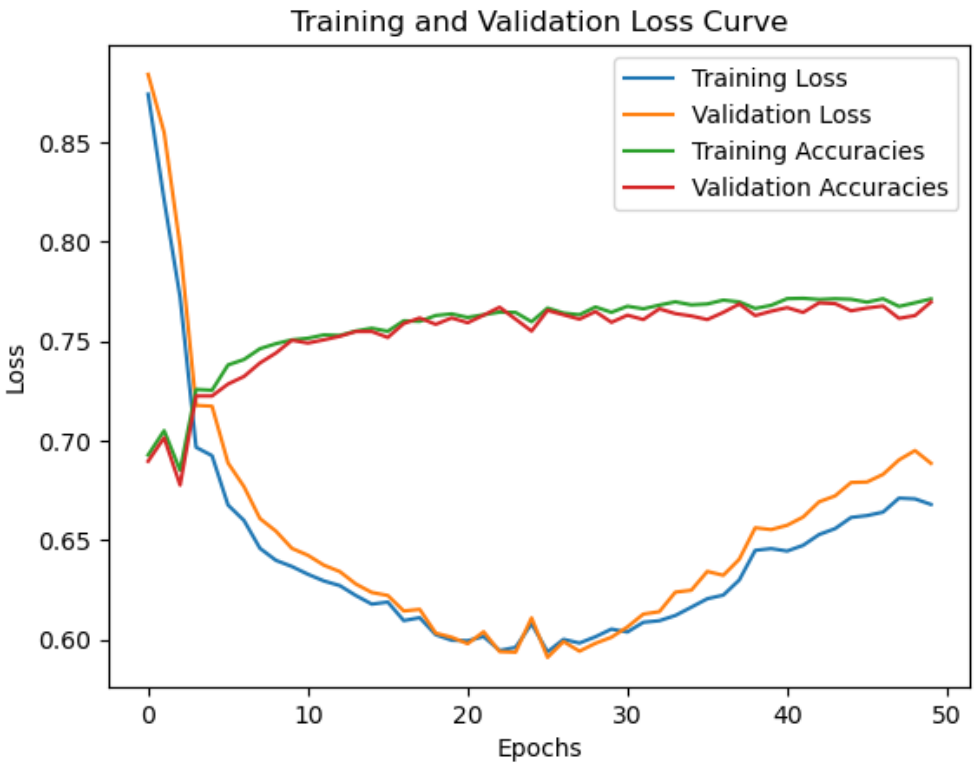


Figure 14: Neural Network Loss Curves

A.8 Hyperparameter Tuning

Hyperparameter Tuning occured throughout the process of implementing the neural network. We tried various hidden layer sizes (e.g. 32, 64, 96, 128) we found that 96 gave us the greatest performance without a large sacrifice on time performance. We experimented with different learning rates before settling on 0.01. We also tested various batch sizes (e.g. 50, 100, 150) before deciding on 50. We found that 50 batches allowed us to properly train the data without approaching overfitting, the larger numbers of batches only performed better on the training data.

A.9 Code Accessibility

The complete code implementation is available through GitHub at github.com/ramafications/Evergreen. For reference, a pdf of the code is also included below.

```
# @title
%pip install ucimlrepo
from ucimlrepo import fetch_ucirepo
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
%pip install scikit-lego
from sklego.linear_model import LADRegression
from sklearn.model_selection import train_test_split
from sklearn import metrics
import seaborn as sns
import numpy as np
import plotly.express as px
import plotly.graph_objects as go
from sklearn.model_selection import KFold
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_validate
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (accuracy_score, confusion_matrix, classification_report,
                             roc_curve, roc_auc_score, f1_score)
from sklearn.preprocessing import label_binarize
from sklearn.metrics import classification_report
from sklearn.decomposition import PCA

```

```
Requirement already satisfied: ucimlrepo in /usr/local/lib/python3.10/dist-packages (0.0.7)
Requirement already satisfied: pandas>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from ucimlrepo) (2.2.2)
Requirement already satisfied: certifi>=2020.12.5 in /usr/local/lib/python3.10/dist-packages (from ucimlrepo) (2024.8.30)
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0->ucimlrepo) (1.2
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0->ucimlr
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0->ucimlrepo) (2024
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0->ucimlrepo) (20
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas>=1.0
Requirement already satisfied: scikit-lego in /usr/local/lib/python3.10/dist-packages (0.9.3)
Requirement already satisfied: narwhals>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from scikit-lego) (1.17.0)
Requirement already satisfied: pandas>=1.1.5 in /usr/local/lib/python3.10/dist-packages (from scikit-lego) (2.2.2)
Requirement already satisfied: scikit-learn>=1.0 in /usr/local/lib/python3.10/dist-packages (from scikit-lego) (1.5.2)
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1.5->scikit-lego) (1
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1.5->scikit
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1.5->scikit-lego) (20
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1.5->scikit-lego) (
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0->scikit-lego)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0->scikit-lego
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0->scik
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas>=1.1
```

```
# @title

# Fetch dataset
covertype = fetch_ucirepo(id=31)

# Get the features DataFrame
X = covertype.data.features

# Print all column names
print("All columns in the dataset:")
for column in X.columns:
    print(column)

# Print the total number of columns
print(f"\nTotal number of columns: {len(X.columns)}")
```

```
All columns in the dataset:
Elevation
Aspect
Slope
Horizontal_Distance_To_Hydrology
Vertical_Distance_To_Hydrology
Horizontal_Distance_To_Roadways
Hillshade_9am
Hillshade_Noon
Hillshade_3pm
Horizontal_Distance_To_Fire_Points
```

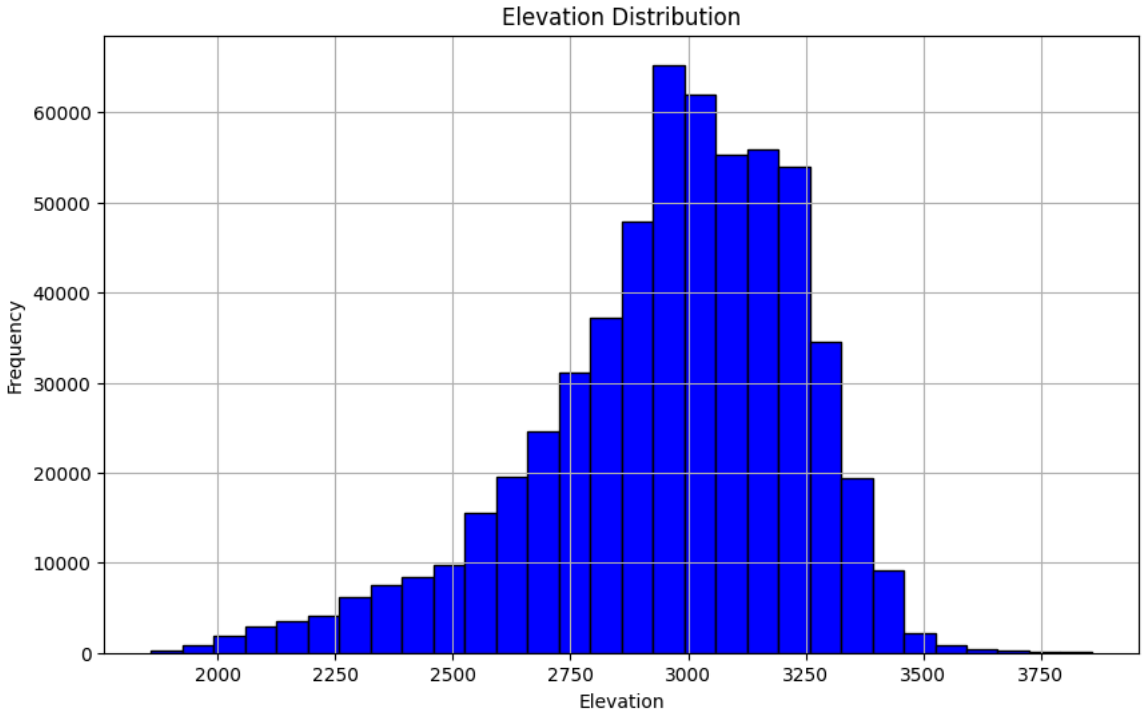
```
Wilderness_Area1
Soil_Type1
Soil_Type2
Soil_Type3
Soil_Type4
Soil_Type5
Soil_Type6
Soil_Type7
Soil_Type8
Soil_Type9
Soil_Type10
Soil_Type11
Soil_Type12
Soil_Type13
Soil_Type14
Soil_Type15
Soil_Type16
Soil_Type17
Soil_Type18
Soil_Type19
Soil_Type20
Soil_Type21
Soil_Type22
Soil_Type23
Soil_Type24
Soil_Type25
Soil_Type26
Soil_Type27
Soil_Type28
Soil_Type29
Soil_Type30
Soil_Type31
Soil_Type32
Soil_Type33
Soil_Type34
Soil_Type35
Soil_Type36
Soil_Type37
Soil_Type38
Soil_Type39
Soil_Type40
Wilderness_Area2
Wilderness_Area3
Wilderness_Area4

Total number of columns: 54
```

```
# @title
# data (as pandas dataframes)
X = covertype.data.features
y = covertype.data.targets

# Convert X and y into a single DataFrame
# Assuming that X and y are already in compatible formats
df = pd.DataFrame(X)
df['Target'] = y

# Display the first few rows of the DataFrame
print(df.head())
```

```
   Elevation  Aspect  Slope  Horizontal_Distance_To_Hydrology  \
0       2596      51      3                               258
1       2590      56      2                               212
2       2804     139      9                               268
3       2785     155     18                               242
4       2595      45      2                               153

   Vertical_Distance_To_Hydrology  Horizontal_Distance_To_Roadways  \
0                               0                              510
1                              -6                              390
2                              65                             3180
3                             118                             3090
4                              -1                              391

   Hillshade_9am  Hillshade_Noon  Hillshade_3pm  \
0            221             232            148
1            220             235            151
2            234             238            135
3            238             238            122
4            220             234            150

   Horizontal_Distance_To_Fire_Points  ...  Soil_Type35  Soil_Type36  \
```

```
0                            6279  ...            0            0
1                            6225  ...            0            0
2                            6121  ...            0            0
3                            6211  ...            0            0
4                            6172  ...            0            0

   Soil_Type37  Soil_Type38  Soil_Type39  Soil_Type40  Wilderness_Area2  \
0            0            0            0            0                 0
1            0            0            0            0                 0
2            0            0            0            0                 0
3            0            0            0            0                 0
4            0            0            0            0                 0

   Wilderness_Area3  Wilderness_Area4  Target
0                 0                 0       5
1                 0                 0       5
2                 0                 0       2
3                 0                 0       2
4                 0                 0       5

[5 rows x 55 columns]
```

```
# @title
import matplotlib.pyplot as plt

# Plot a histogram of the Elevation column
plt.figure(figsize=(10, 6))
plt.hist(df['Elevation'], bins=30, color='blue', edgecolor='black')
plt.title('Elevation Distribution')
plt.xlabel('Elevation')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```




```
# @title
## Chack that there is no missingness ##
# Calculate the percentage of missing values for each column
missingness = df.isnull().mean() * 100

# Display columns with missingness
print(f"Missingness for each feature:")
print(missingness[missingness > 0])
```

```
Missingness for each feature:
Series([], dtype: float64)
```

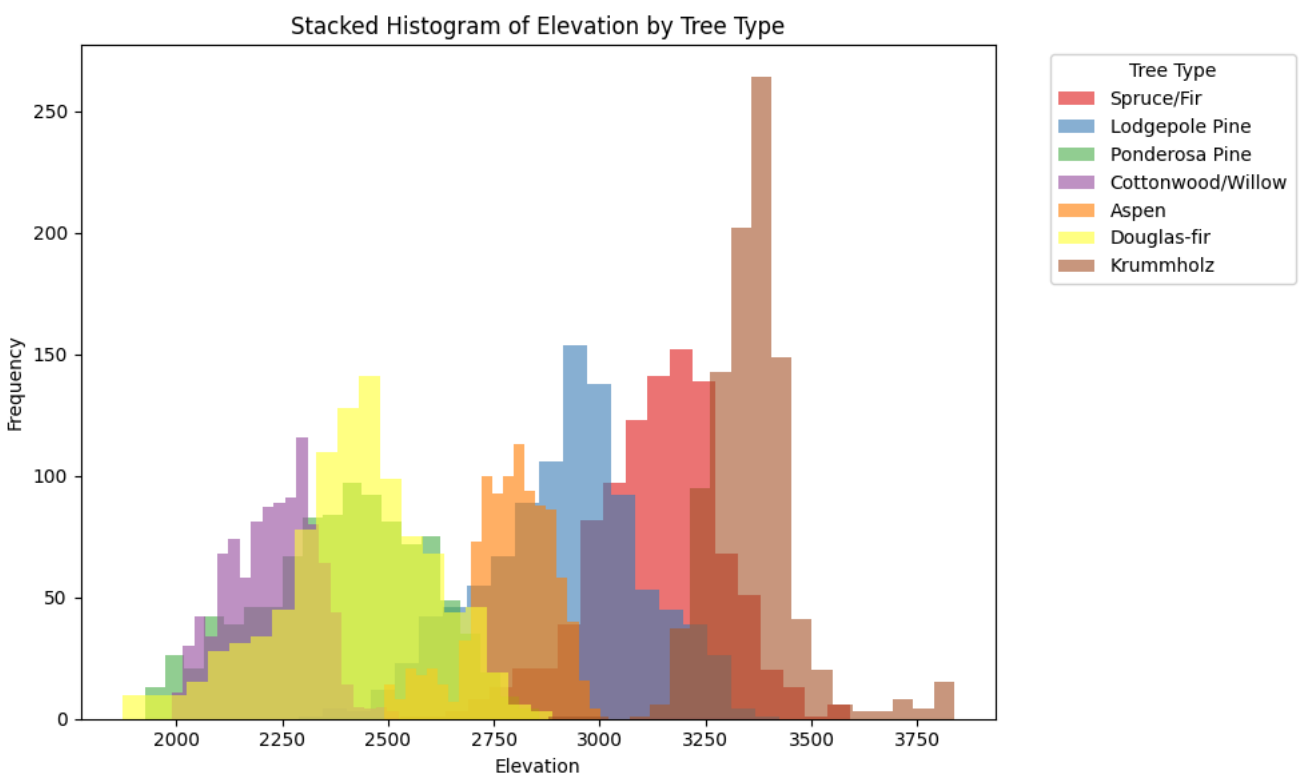
```
# @title
# Taking a stratified sample in order to visualize distrubutions more clearly
```

```
strat_samp = df.groupby('Target').sample(1000)
print(strat_samp)
```

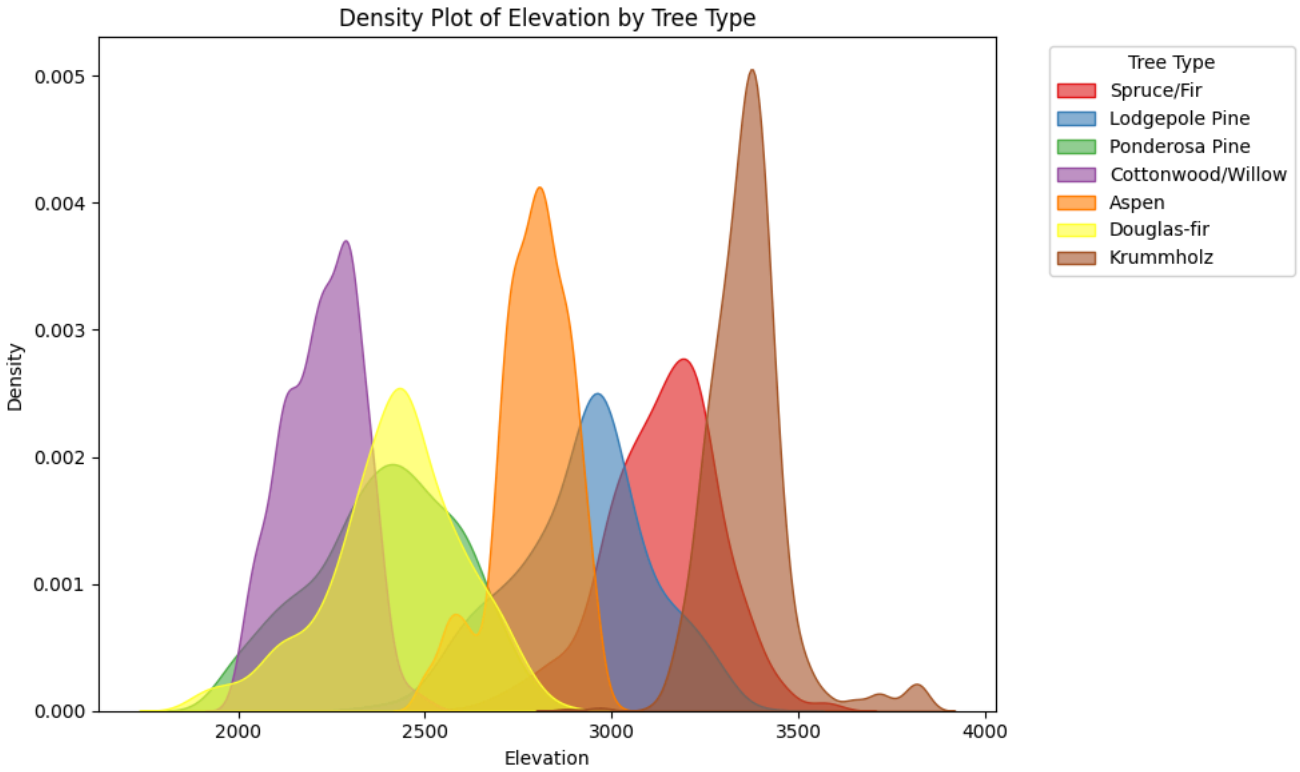
```
        Elevation  Aspect  Slope  Horizontal_Distance_To_Hydrology  \
58807        3004      90      1                               218
450530       3197      13     13                               553
206690       3251     277      6                               330
156976       3004       0      0                                85
102853       3050      35     18                                30
...           ...     ...    ...                               ...
469021       3255     352     13                                30
382814       3401     182      8                               979
459126       3318     321     12                               569
537563       3580      45      6                               379
459069       3395     179     11                                60

        Vertical_Distance_To_Hydrology  Horizontal_Distance_To_Roadways  \
58807                               32                             5210
450530                              68                             2336
206690                             108                             4709
156976                              19                             5332
102853                               0                             3900
...                                ...                              ...
469021                               4                             3108
382814                             162                             2984
459126                              63                             3422
537563                             148                             1724
459069                              -1                             4250

        Hillshade_9am  Hillshade_Noon  Hillshade_3pm  \
58807             221             236            151
450530            207             212            141
206690            204             241            176
156976            218             237            156
102853            215             197            112
...               ...             ...            ...
469021            196             217            158
382814            223             245            156
459126            188             228            179
537563            222             227            141
459069            225             247            153

        Horizontal_Distance_To_Fire_Points  ...  Soil_Type35  Soil_Type36  \
58807                                 6243  ...            0            0
450530                                3371  ...            0            0
206690                                 819  ...            0            0
156976                                2415  ...            0            0
102853                                3900  ...            0            0
...                                    ...  ...          ...          ...
469021                                2594  ...            0            0
382814                                1687  ...            0            0
459126                                3487  ...            0            0
537563                                1549  ...            0            0
459069                                4079  ...            0            0

        Soil_Type37  Soil_Type38  Soil_Type39  Soil_Type40  Wilderness_Area2  \
58807             0            0            0            0                 0
450530            0            0            0            0                 0
206690            0            0            0            0                 0
156976            0            0            0            0                 0
102853            0            0            0            0                 0
```

```
# @title
# Visualize how the elevation is split between each tree type
# Set up plot
plt.figure(figsize=(10, 6))
bins = 20
colors = sns.color_palette("Set1", 7)

# Rename variables
tree_names = {
    1: "Spruce/Fir",
    2: "Lodgepole Pine",
    3: "Ponderosa Pine",
    4: "Cottonwood/Willow",
    5: "Aspen",
    6: "Douglas-fir",
    7: "Krummholz"
}

# Plot each histogram
```

```
for i in range(1, 8):
    subset = strat_samp[strat_samp['Target'] == i]
    plt.hist(subset['Elevation'], bins=bins, alpha=0.6, label=tree_names[i], color=colors[i-1], stacked=True)

# Add labels/title and plot
plt.title("Stacked Histogram of Elevation by Tree Type")
plt.xlabel("Elevation")
plt.ylabel("Frequency")
plt.legend(title='Tree Type', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()
```



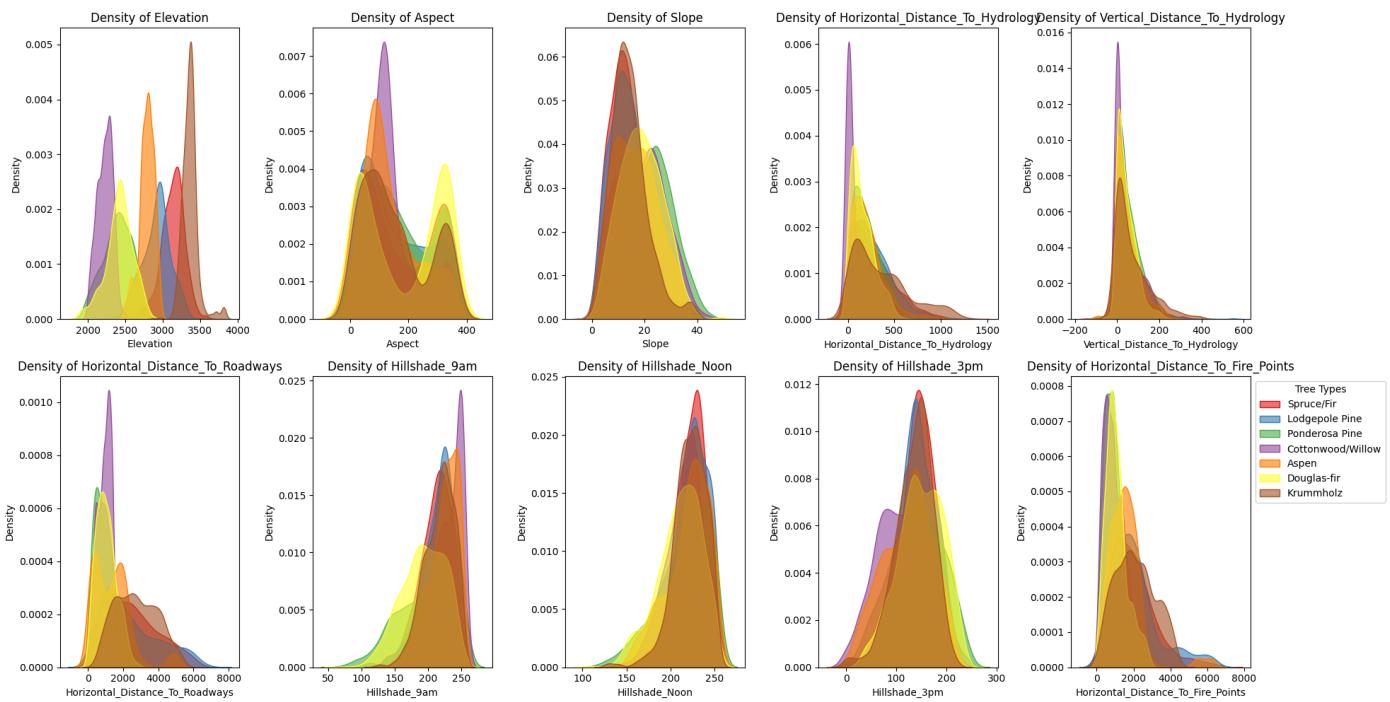

```
# @title
# Can change to a density curve for clearer visualization
# Set up plot
plt.figure(figsize=(10, 6))
bins = 20  # You can adjust this, but KDE plots don't require bins like histograms
colors = sns.color_palette("Set1", 7)

# Rename variables
tree_names = {
    1: "Spruce/Fir",
    2: "Lodgepole Pine",
    3: "Ponderosa Pine",
    4: "Cottonwood/Willow",
    5: "Aspen",
    6: "Douglas-fir",
    7: "Krummholz"
}

# Plot each density curve
for i in range(1, 8):
    subset = strat_samp[strat_samp['Target'] == i]
    sns.kdeplot(subset['Elevation'], fill=True, color=colors[i-1], label=tree_names[i], alpha=0.6)

# Add labels/title and plot
plt.title("Density Plot of Elevation by Tree Type")
plt.xlabel("Elevation")
plt.ylabel("Density")
plt.legend(title='Tree Type', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()
```



We can see that Spruce/Fir and Lodgepole Pines occur at elevations primarily between 2500-3500, Ponderosa Pines and Douglass-fir occur at lower elevations (between 2000-2750)

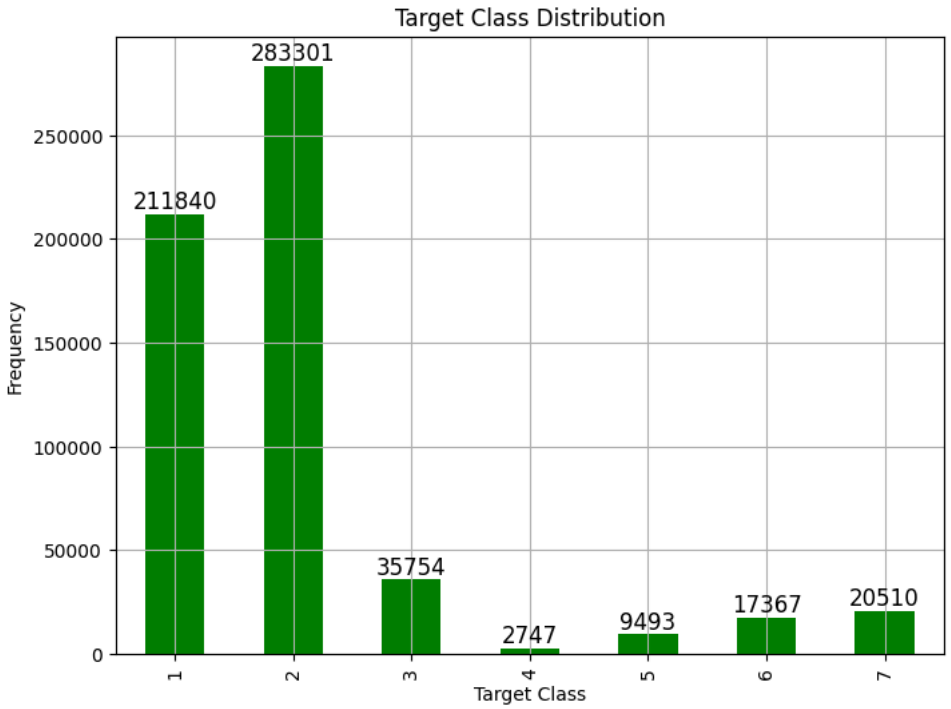
```
# @title
# Rename variables
tree_names = {
    1: "Spruce/Fir",
    2: "Lodgepole Pine",
    3: "Ponderosa Pine",
    4: "Cottonwood/Willow",
    5: "Aspen",
    6: "Douglas-fir",
    7: "Krummholz"
}

# Define the variables
variables = ['Elevation', 'Aspect', 'Slope', 'Horizontal_Distance_To_Hydrology', 'Vertical_Distance_To_Hydrology',
             'Horizontal_Distance_To_Roadways', 'Hillshade_9am', 'Hillshade_Noon', 'Hillshade_3pm', 'Horizontal_Distance_To_F

# Set up the subplot grid (2 columns and 5 rows)
fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(18, 10))
axes = axes.flatten()
colors = sns.color_palette("Set1", 7)

# Plot KDEs for each variable
for i, variable in enumerate(variables):
    ax = axes[i]

    for tree_type in range(1, 8):
        # Filter the data for the specific tree type
        subset = strat_samp[strat_samp['Target'] == tree_type]

        # Plot the KDE for the current tree type's values of the variable
        sns.kdeplot(subset[variable], ax=ax, fill=True, color=colors[tree_type-1], label=tree_names[tree_type], alpha=0.6)

    # Customize the plot for this variable
    ax.set_title(f"Density of {variable}")
    ax.set_xlabel(variable)
    ax.set_ylabel("Density")

# Adjust layout for better spacing
plt.tight_layout()
ax.legend(title='Tree Types', loc='upper left', bbox_to_anchor=(1, 1))
```

```
41
42 # Show the plot
43 plt.show()
```



Visualizing the different variables it appears that Elevation, Horizontal Distance to Roadways and Fire Points may be the easiest to use for a predictor model because they appear to be the most spread out of all the variables

```
1 # @title
2 # Visualize the Target Class Distribution
3 plt.figure(figsize=(8, 6))
4 ax = df['Target'].value_counts().sort_index().plot(kind='bar', color='green')
5
6 # Add the number of trees in each category as text on top of the bars
7 for i, count in enumerate(df['Target'].value_counts().sort_index()):
8     ax.text(i, count + 0.1, str(count), ha='center', va='bottom', fontsize=12)
9
10 plt.title('Target Class Distribution')
11 plt.xlabel('Target Class')
12 plt.ylabel('Frequency')
13 plt.grid(True)
14 plt.show()
```



There are significantly more recorded instances of Spruce/Fir and Lodgepole Pines as opposed to the other tree types

```
# @title
# Filter columns that are related to Soil Type
soil_columns = [col for col in df.columns if 'Soil_Type' in col]

# Display the Soil Type columns
print("Soil Types Columns:")
print(soil_columns)

# Now, let's print the first few rows of the Soil Type data
soil_data = df[soil_columns]
print("\nFirst 5 rows of Soil Type Data:")
print(soil_data)
```

```
Soil Types Columns:
['Soil_Type1', 'Soil_Type2', 'Soil_Type3', 'Soil_Type4', 'Soil_Type5', 'Soil_Type6', 'Soil_Type7', 'Soil_Type8', 'Soil_Type9

First 5 rows of Soil Type Data:
        Soil_Type1  Soil_Type2  Soil_Type3  Soil_Type4  Soil_Type5  \
0                0           0           0           0           0
1                0           0           0           0           0
2                0           0           0           0           0
3                0           0           0           0           0
4                0           0           0           0           0
...            ...         ...         ...         ...         ...
581007           0           1           0           0           0
581008           0           1           0           0           0
581009           0           1           0           0           0
581010           0           1           0           0           0
581011           0           1           0           0           0

        Soil_Type6  Soil_Type7  Soil_Type8  Soil_Type9  Soil_Type10  ...  \
0                0           0           0           0            0  ...
1                0           0           0           0            0  ...
2                0           0           0           0            0  ...
3                0           0           0           0            0  ...
4                0           0           0           0            0  ...
...            ...         ...         ...         ...          ...  ...
581007           0           0           0           0            0  ...
581008           0           0           0           0            0  ...
581009           0           0           0           0            0  ...
581010           0           0           0           0            0  ...
581011           0           0           0           0            0  ...

        Soil_Type31  Soil_Type32  Soil_Type33  Soil_Type34  Soil_Type35  \
0                 0            0            0            0            0
1                 0            0            0            0            0
```

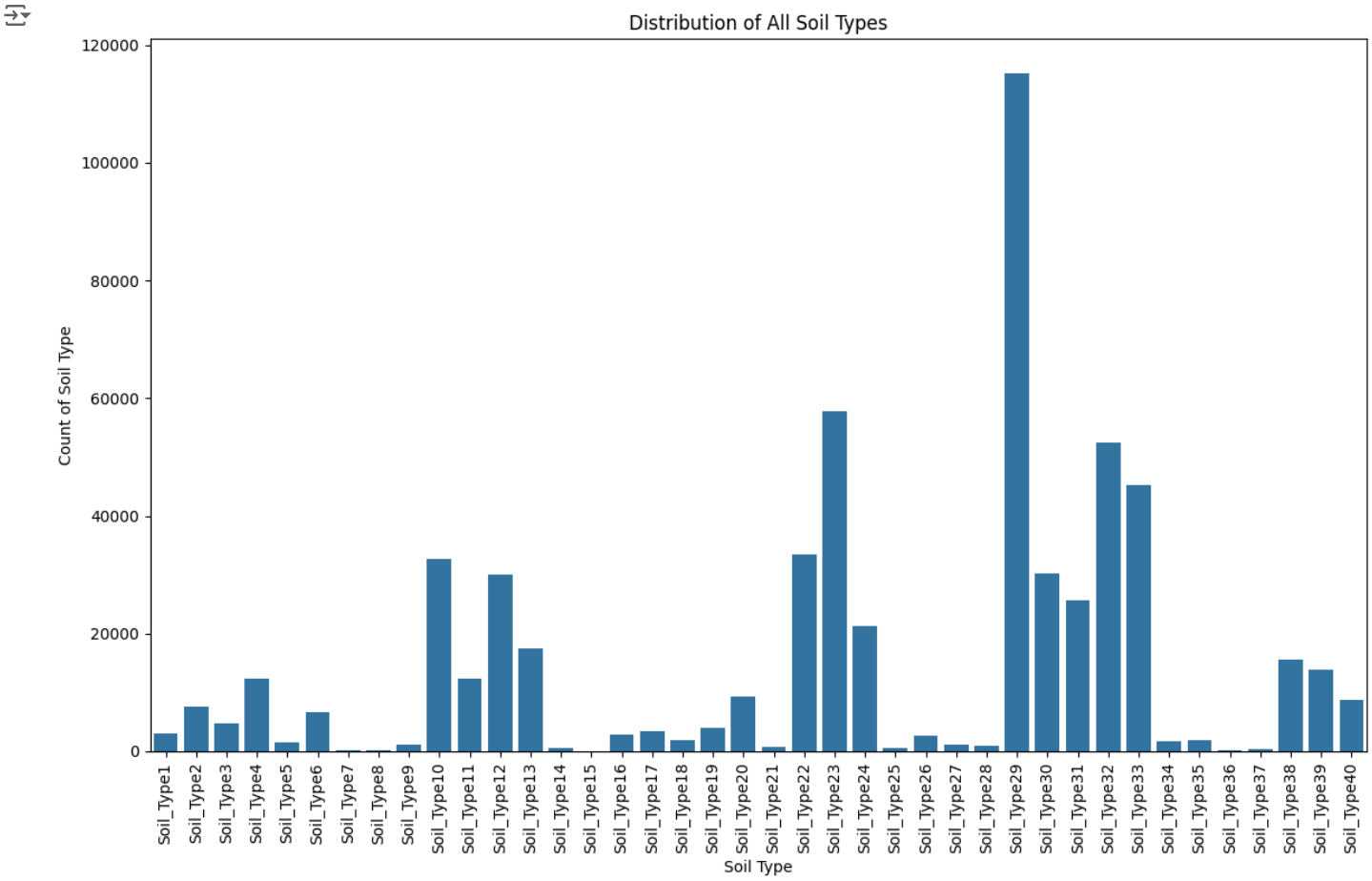
```
2                0            0            0            0            0
3                0            0            0            0            0
4                0            0            0            0            0
...            ...          ...          ...          ...          ...
581007           0            0            0            0            0
581008           0            0            0            0            0
581009           0            0            0            0            0
581010           0            0            0            0            0
581011           0            0            0            0            0

        Soil_Type36  Soil_Type37  Soil_Type38  Soil_Type39  Soil_Type40
0                0            0            0            0            0
1                0            0            0            0            0
2                0            0            0            0            0
3                0            0            0            0            0
4                0            0            0            0            0
...            ...          ...          ...          ...          ...
581007           0            0            0            0            0
581008           0            0            0            0            0
581009           0            0            0            0            0
581010           0            0            0            0            0
581011           0            0            0            0            0

[581012 rows x 40 columns]
```

```python
# @title
# Filter columns that are related to Soil Type
soil_columns = [col for col in X.columns if 'Soil_Type' in col]

# Count the number of 1s for each soil type
soil_counts = X[soil_columns].sum().sort_values(ascending=False)

# Create a DataFrame with the counts
soil_counts_df = pd.DataFrame({'Soil Type': soil_counts.index, 'Count': soil_counts.values})

# Display the counts
print(soil_counts_df)

# Check for any soil types with zero occurrences
zero_soil_types = soil_counts[soil_counts == 0]
if not zero_soil_types.empty:
    print("\nSoil types with zero occurrences:")
    print(zero_soil_types)
else:
    print("\nAll soil types have at least one occurrence.")
```

```
       Soil Type   Count
0   Soil_Type29  115247
1   Soil_Type23   57752
2   Soil_Type32   52519
3   Soil_Type33   45154
4   Soil_Type22   33373
5   Soil_Type10   32634
6   Soil_Type30   30170
7   Soil_Type12   29971
8   Soil_Type31   25666
9   Soil_Type24   21278
10  Soil_Type13   17431
11  Soil_Type38   15573
12  Soil_Type39   13806
13  Soil_Type11   12410
14   Soil_Type4   12396
15  Soil_Type20    9259
16  Soil_Type40    8750
17   Soil_Type2    7525
18   Soil_Type6    6575
19   Soil_Type3    4823
20  Soil_Type19    4021
21  Soil_Type17    3422
22   Soil_Type1    3031
23  Soil_Type16    2845
24  Soil_Type26    2589
25  Soil_Type18    1899
26  Soil_Type35    1891
27  Soil_Type34    1611
28   Soil_Type5    1597
29   Soil_Type9    1147
30  Soil_Type27    1086
31  Soil_Type28     946
32  Soil_Type21     838
33  Soil_Type14     599
34  Soil_Type25     474
```

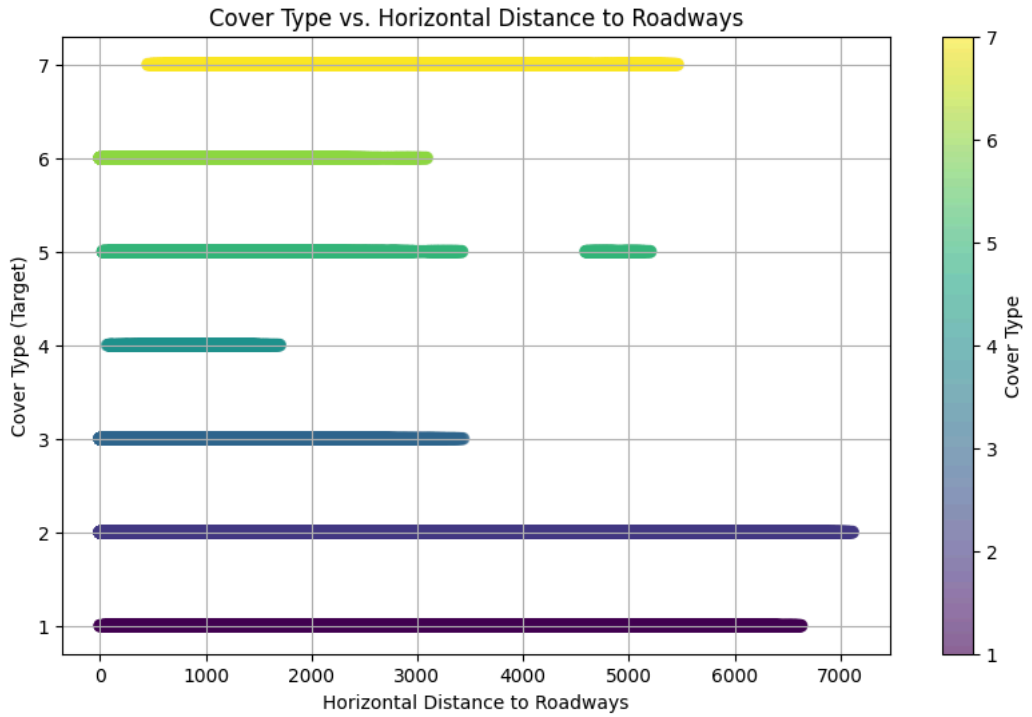
```
35  Soil_Type37     298
36   Soil_Type8     179
37  Soil_Type36     119
38   Soil_Type7     105
39  Soil_Type15       3

All soil types have at least one occurrence.
```

```
# @title
import seaborn as sns
# Filter columns that are related to Soil Type
soil_columns = [col for col in df.columns if 'Soil_Type' in col]

# Summing up the occurrences of each soil type across all rows
soil_counts = df[soil_columns].sum()

# Plotting the bar graph for all soil types
plt.figure(figsize=(12,8))
sns.barplot(x=soil_counts.index, y=soil_counts.values)
plt.title("Distribution of All Soil Types")
plt.xlabel("Soil Type")
plt.ylabel("Count of Soil Type")
plt.xticks(rotation=90)
plt.tight_layout()

# Show the plot
plt.show()

```



Soil types where there are no observations in the data (7, 8, 14, 15, 21, 25, 26, 27, 28, 34, 35, 36, 37, etc.) we will data clean to exclude. We will use L1 and L2 regression, and determine which gives the line of best fit, in order to analyze our data.

```
# @title
plt.figure(figsize=(10, 6))
plt.scatter(df['Horizontal_Distance_To_Roadways'], df['Target'], alpha=0.6, c=df['Target'], cmap='viridis')
plt.title('Cover Type vs. Horizontal Distance to Roadways')
plt.xlabel('Horizontal Distance to Roadways')
plt.ylabel('Cover Type (Target)')
plt.colorbar(label='Cover Type')
plt.grid(True)
plt.show()
```




```
# @title
%pip install scikit-lego
%pip install sklego
```

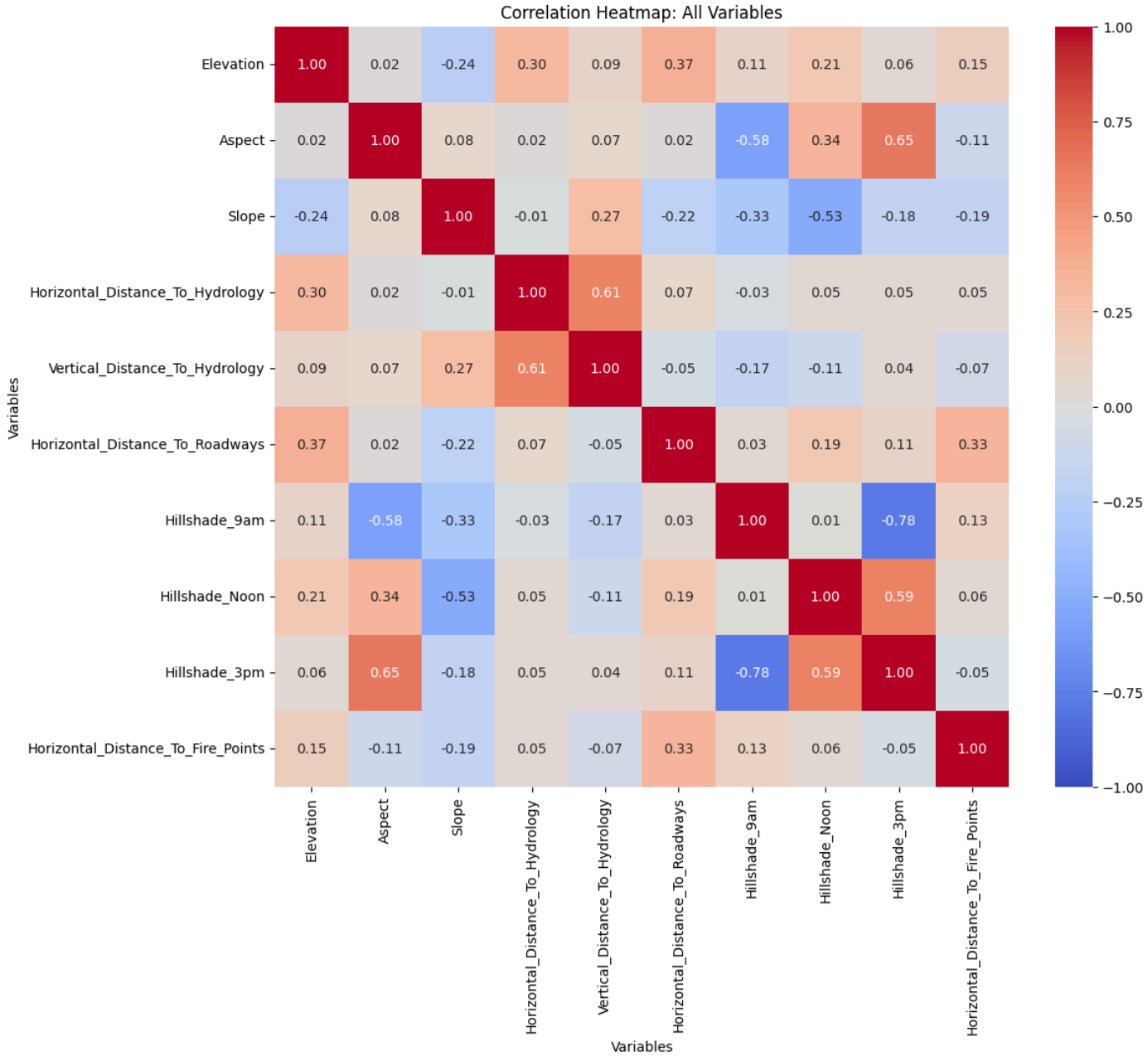
```
Requirement already satisfied: scikit-lego in /usr/local/lib/python3.10/dist-packages (0.9.3)
Requirement already satisfied: narwhals>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from scikit-lego) (1.17.0)
Requirement already satisfied: pandas>=1.1.5 in /usr/local/lib/python3.10/dist-packages (from scikit-lego) (2.2.2)
Requirement already satisfied: scikit-learn>=1.0 in /usr/local/lib/python3.10/dist-packages (from scikit-lego) (1.5.2)
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1.5->scikit-lego) (1
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1.5->scikit
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1.5->scikit-lego) (20
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1.5->scikit-lego) (
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0->scikit-lego)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0->scikit-lego
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0->scik
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas>=1.1
Collecting sklego
  Downloading sklego-0.0-py2.py3-none-any.whl.metadata (375 bytes)
Requirement already satisfied: scikit-lego in /usr/local/lib/python3.10/dist-packages (from sklego) (0.9.3)
Requirement already satisfied: narwhals>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from scikit-lego->sklego) (1.17.0
Requirement already satisfied: pandas>=1.1.5 in /usr/local/lib/python3.10/dist-packages (from scikit-lego->sklego) (2.2.2)
Requirement already satisfied: scikit-learn>=1.0 in /usr/local/lib/python3.10/dist-packages (from scikit-lego->sklego) (1.5.
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1.5->scikit-lego->sk
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1.5->scikit
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1.5->scikit-lego->skl
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.1.5->scikit-lego->s
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0->scikit-lego-
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0->scikit-lego
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=1.0->scik
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas>=1.1
Downloading sklego-0.0-py2.py3-none-any.whl (1.1 kB)
Installing collected packages: sklego
Successfully installed sklego-0.0
```

```
# @title
# First, split the data into training and temp (which will later be split into validation and testing)
train_data, temp_data = train_test_split(df, test_size=0.4, random_state=42) # the random state will ensure we will all start

```

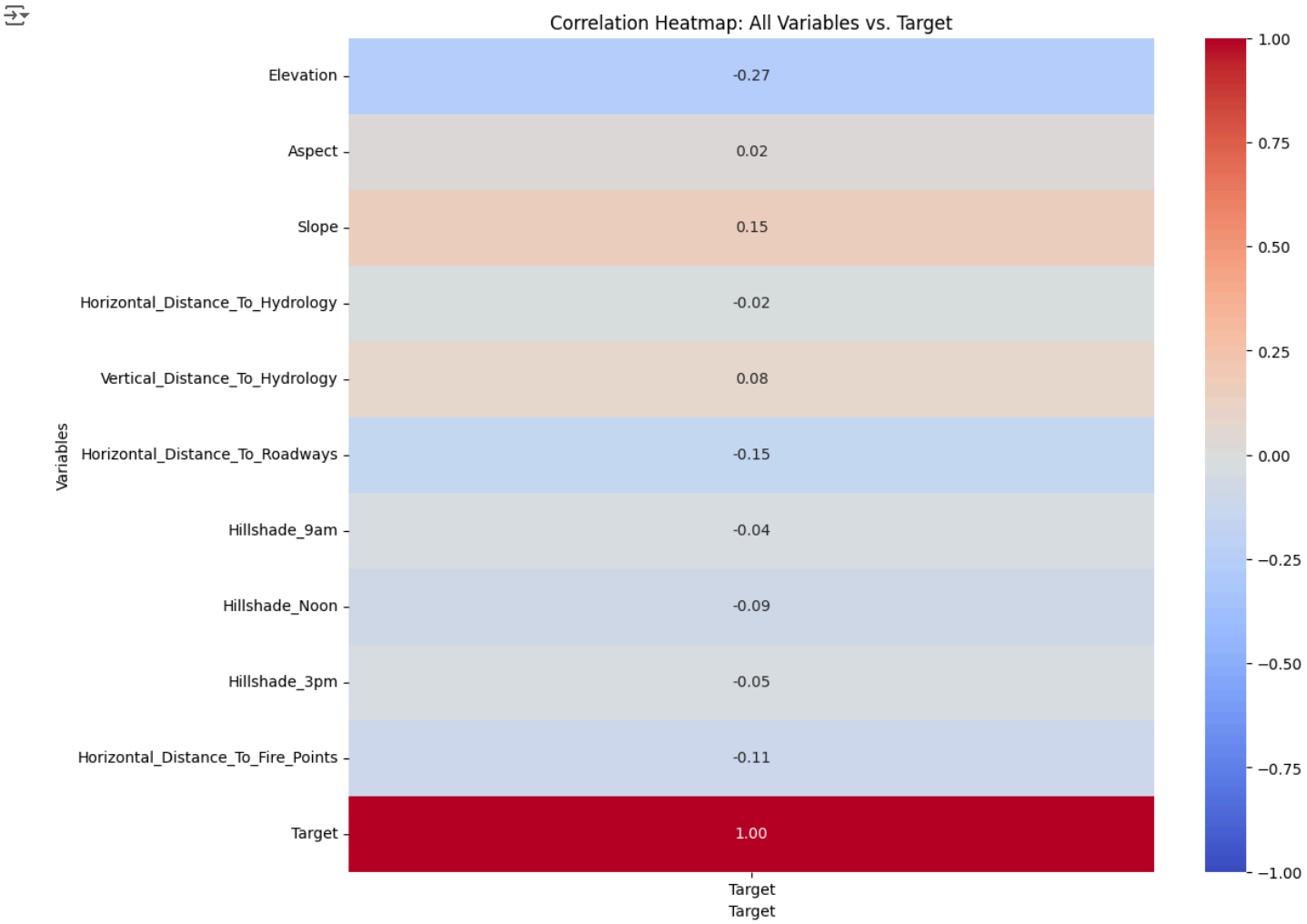
```
# Now split the temp data into validation and testing sets
val_data, test_data = train_test_split(temp_data, test_size=0.5, random_state=42)

# Check the sizes of each set
print(f'Training set size: {train_data.shape[0]}')
print(f'Validation set size: {val_data.shape[0]}')
print(f'Test set size: {test_data.shape[0]}')
```

```
Training set size: 348607
Validation set size: 116202
Test set size: 116203
```

```
# @title
# Since we are predicting Cover Type (Target), we need to figure out what variable to use and we want to make sure to check c

# Calculate the full correlation matrix for the dataset (dropping the binary variables)
correlation = train_data.drop(
    train_data.filter(like='Soil_Type').columns.append(train_data.filter(like='Wilderness_Area').columns.append(train_data.fi
    axis=1
)

correlation_matrix = correlation.corr()

# Visualize the entire correlation matrix
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', vmin=-1, vmax=1, fmt=".2f")
plt.title('Correlation Heatmap: All Variables')
plt.xlabel('Variables')
plt.ylabel('Variables')
plt.show()
```


Correlation Heatmap: All Variables
Elevation 1.00 0.02 -0.24 0.30 0.09 0.37 0.11 0.21 0.06 0.15
Aspect 0.02 1.00 0.08 0.02 0.07 0.02 -0.58 0.34 0.65 -0.11
Slope -0.24 0.08 1.00 -0.01 0.27 -0.22 -0.33 -0.53 -0.18 -0.19
Horizontal_Distance_To_Hydrology 0.30 0.02 -0.01 1.00 0.61 0.07 -0.03 0.05 0.05 0.05
Vertical_Distance_To_Hydrology 0.09 0.07 0.27 0.61 1.00 -0.05 -0.17 -0.11 0.04 -0.07
Horizontal_Distance_To_Roadways 0.37 0.02 -0.22 0.07 -0.05 1.00 0.03 0.19 0.11 0.33
Hillshade_9am 0.11 -0.58 -0.33 -0.03 -0.17 0.03 1.00 0.01 -0.78 0.13
Hillshade_Noon 0.21 0.34 -0.53 0.05 -0.11 0.19 0.01 1.00 0.59 0.06
Hillshade_3pm 0.06 0.65 -0.18 0.05 0.04 0.11 -0.78 0.59 1.00 -0.05
Horizontal_Distance_To_Fire_Points 0.15 -0.11 -0.19 0.05 -0.07 0.33 0.13 0.06 -0.05 1.00
Variables
Elevation
Aspect
Slope
Horizontal_Distance_To_Hydrology
Vertical_Distance_To_Hydrology
Horizontal_Distance_To_Roadways
Hillshade_9am
Hillshade_Noon
Hillshade_3pm
Horizontal_Distance_To_Fire_Points
Variables
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

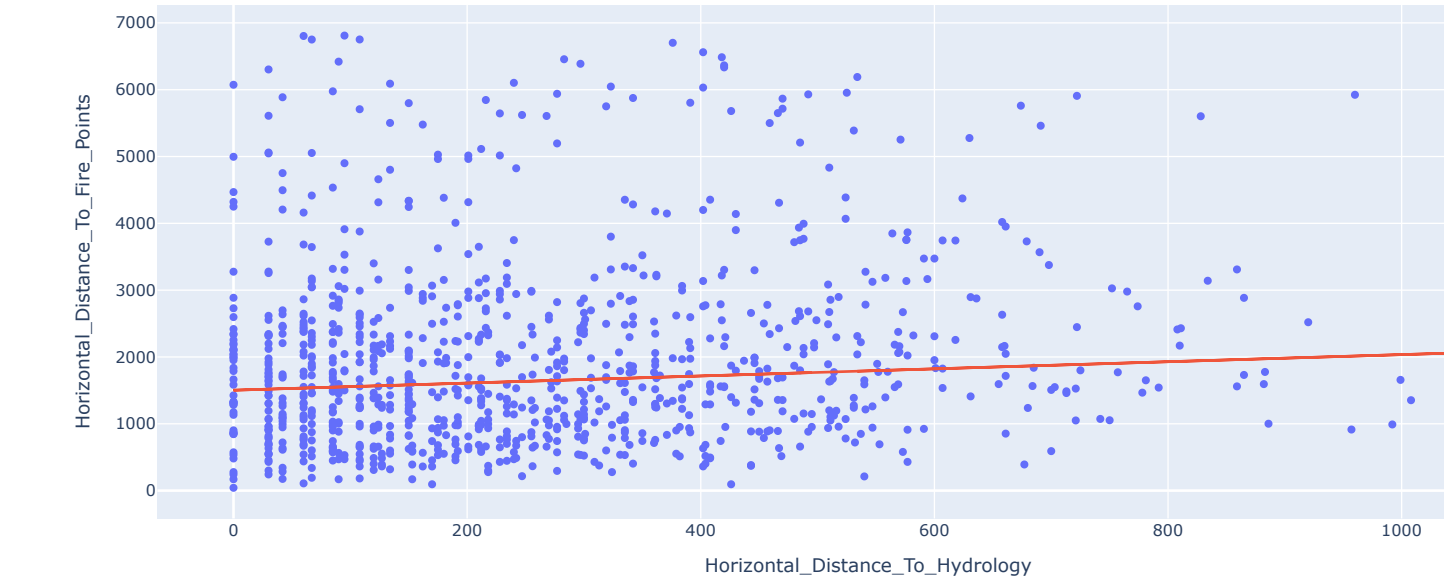
```
# @title
# Since we are predicting Cover Type (Target), we need to figure out what variable to use

# Calculate the full correlation matrix for the dataset (dropping the binary variables)
correlation = train_data.drop(
    train_data.filter(like='Soil_Type').columns.append(train_data.filter(like='Wilderness_Area').columns),
    axis=1
)
correlation_matrix = correlation.corr()
target_corr = correlation_matrix[['Target']]

# Visualize the entire correlation matrix
plt.figure(figsize=(12, 10))
sns.heatmap(target_corr, annot=True, cmap='coolwarm', vmin=-1, vmax=1, fmt=".2f")
plt.title('Correlation Heatmap: All Variables vs. Target')
plt.xlabel('Target')
plt.ylabel('Variables')
plt.show()
```



Elevation is the most highley corrrelated followed by slope, horizontal distance to roadways and horizontal distance to fire points, which we predicted from looking at the histograms above.

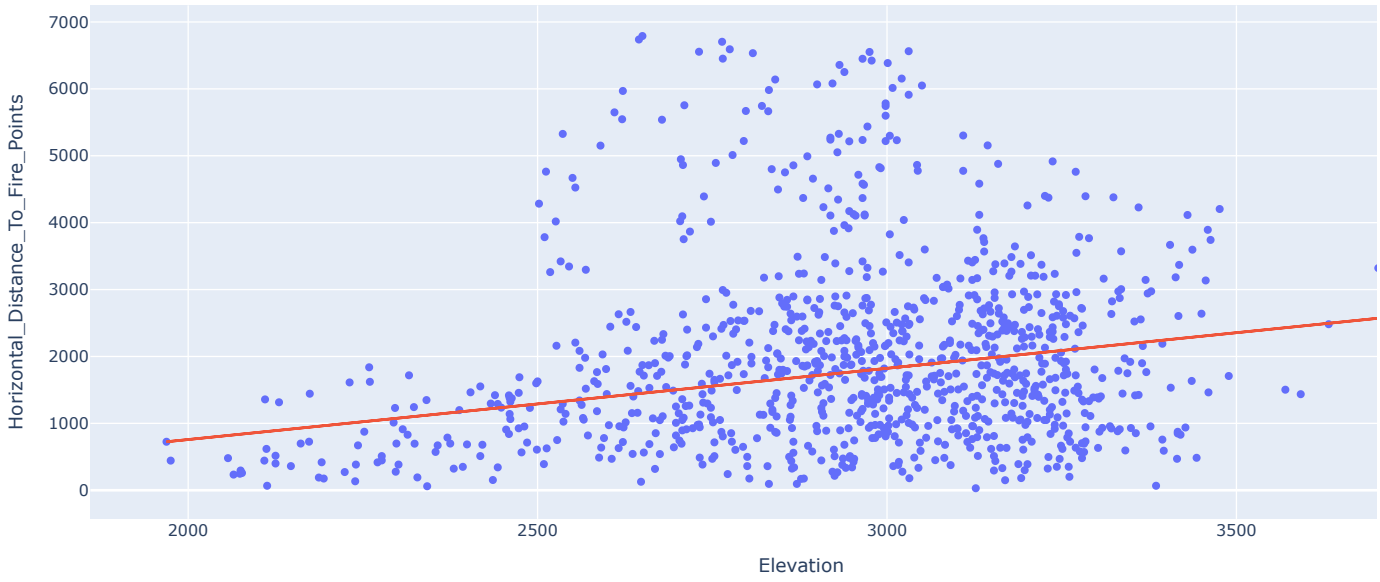
```
# @title
# Check for colinearity amongst the x variables

pd.set_option('display.max_columns', None)
pd.options.display.max_colwidth = 500
pd.options.display.max_rows = 100

```

```
# Sample the data
sample_df = train_data.sample(n=1000, random_state=77)

# Fit the LAD regression model
lad_fit = LADRegression()
lad_fit.fit(X=np.array(sample_df["Horizontal_Distance_To_Hydrology"]).reshape(-1, 1),
            y=sample_df["Horizontal_Distance_To_Fire_Points"])

# Get the intercept and coefficient
intercept = lad_fit.intercept_
coef = lad_fit.coef_[0]

# Calculate predicted values
sample_df['predicted'] = intercept + coef * sample_df["Horizontal_Distance_To_Hydrology"]

# Plot the LAD fitted line on top of the scatterplot
fig = px.scatter(sample_df, x="Horizontal_Distance_To_Hydrology", y="Horizontal_Distance_To_Fire_Points")
fig.add_trace(
    go.Scatter(
        x=sample_df["Horizontal_Distance_To_Hydrology"],
        y=sample_df['predicted'],
        mode='lines',
        name='LAD Fitted Line'
    )
)

fig.show()
```

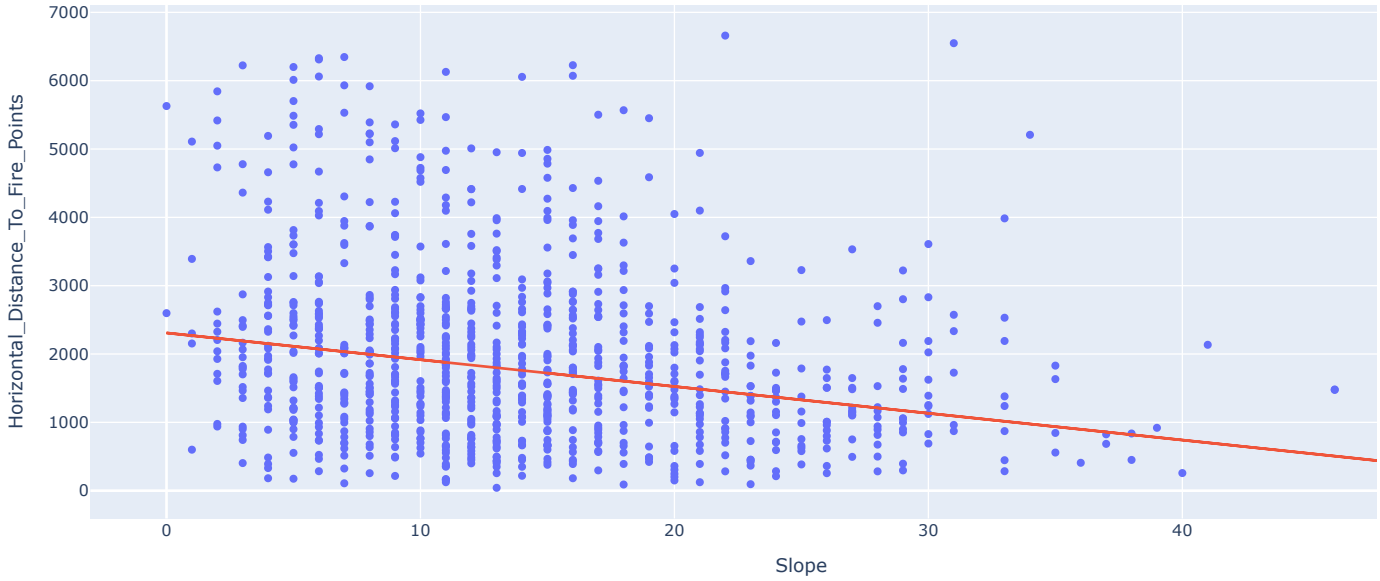



```
# @title
# Sample the data
sample_df = train_data.sample(n=1000, random_state=19)

# Fit the LAD regression model
lad_fit = LADRegression()
lad_fit.fit(X=np.array(sample_df["Elevation"]).reshape(-1, 1),
            y=sample_df["Horizontal_Distance_To_Fire_Points"])

# Get the intercept and coefficient
intercept = lad_fit.intercept_
coef = lad_fit.coef_[0]

# Calculate predicted values
sample_df['predicted'] = intercept + coef * sample_df["Elevation"]

# Plot the LAD fitted line on top of the scatterplot
fig = px.scatter(sample_df, x="Elevation", y="Horizontal_Distance_To_Fire_Points")
fig.add_trace(
    go.Scatter(
```

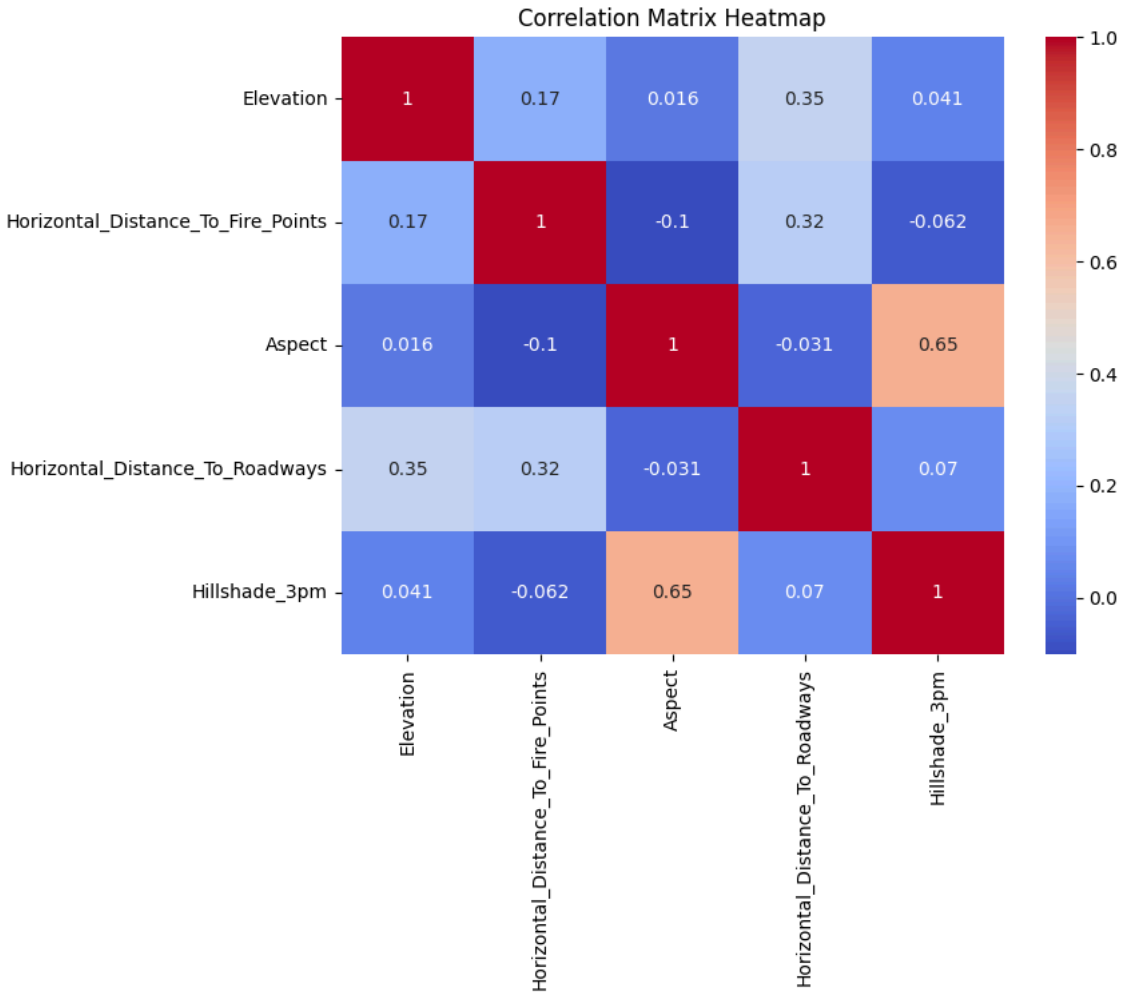
```
        x=sample_df["Elevation"],
        y=sample_df['predicted'],
        mode='lines',
        name='LAD Fitted Line'
    )
)

fig.show()
```




```
# @title
pd.set_option('display.max_columns', None)
pd.options.display.max_colwidth = 500
pd.options.display.max_rows = 100

# Sample the data
sample_df = train_data.sample(n=1000, random_state=42)

# Fit the LAD regression model
lad_fit = LADRegression()
lad_fit.fit(X=np.array(sample_df["Slope"]).reshape(-1, 1),
            y=sample_df["Horizontal_Distance_To_Fire_Points"])

# Get the intercept and coefficient
intercept = lad_fit.intercept_
coef = lad_fit.coef_[0]

# Calculate predicted values
sample_df['predicted'] = intercept + coef * sample_df["Slope"]

# Plot the LAD fitted line on top of the scatterplot
fig = px.scatter(sample_df, x="Slope", y="Horizontal_Distance_To_Fire_Points")
fig.add_trace(
    go.Scatter(
        x=sample_df["Slope"],
        y=sample_df['predicted'],
        mode='lines',
        name='LAD Fitted Line'
    )
)

fig.show()
```




```
# @title
multi = sample_df[["Elevation", "Horizontal_Distance_To_Fire_Points", "Aspect", "Horizontal_Distance_To_Roadways", "Hillshade

# Calculate correlation matrix
corr_matrix = np.corrcoef(np.transpose(multi))  # Transpose the dataframe for np.corrcoef to work

# Create a heatmap
plt.figure(figsize=(8, 6))  # Set the figure size
sns.heatmap(corr_matrix, annot=True, cmap="coolwarm", xticklabels=multi.columns, yticklabels=multi.columns, cbar=True)

# Add labels and title
plt.title("Correlation Matrix Heatmap")
plt.show()
```



Doesn't seem to be colinear, which is good

```
# @title
X = multi  # Independent variables
y = sample_df['Target']  # Dependent variable (target)

# Split into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Create the linear regression model and fit it to the data
model = LinearRegression()
model.fit(X_train_scaled, y_train)

# Get the coefficients and intercept
print("Coefficients:", model.coef_)
print("Intercept:", model.intercept_)

# Make predictions on the test data
y_pred = model.predict(X_test_scaled)

# Evaluate the model
r2 = r2_score(y_test, y_pred)
print("R-squared:", r2)

mae = mean_absolute_error(y_test, y_pred)
print("Mean Absolute Error:", mae)

mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

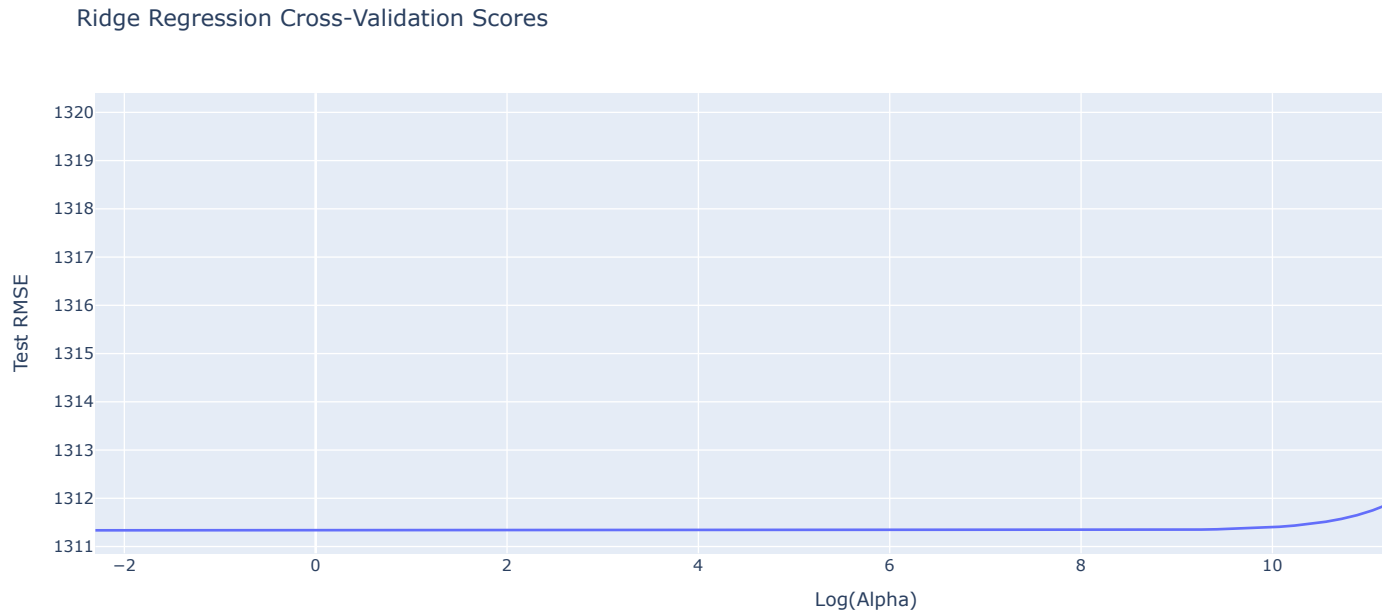
```

rmse = np.sqrt(mse)
print("Root Mean Squared Error:", rmse)
```

```
Coefficients: [-0.32261429 -0.07589237  0.18087524 -0.09621897 -0.16179475]
Intercept: 2.0999999999999996
R-squared: 0.20280055571648847
Mean Absolute Error: 0.6836325468180605
Mean Squared Error: 1.2594954020235196
Root Mean Squared Error: 1.1222724277213263
```

```
# @title
# Define features and target variable
X = train_data[['Elevation']]  # can include more features here later
y = train_data['Horizontal_Distance_To_Fire_Points']

# Standardize the features
scaler = StandardScaler()
X_std = scaler.fit_transform(X)

# Define the alpha values to test
alphas = np.logspace(-1, 6, 100)

# Create a list to store the cross-validation scores
ridge_cv_scores = []

# Loop to compute the cross-validation score for each alpha value
for alpha in alphas:
    ridge = Ridge(alpha=alpha)
    ridge_cv = cross_validate(estimator=ridge,
                              X=X_std,
                              y=y,
                              cv=10,
                              scoring='neg_root_mean_squared_error')
    # Append the results
    ridge_cv_scores.append({
        'alpha': alpha,
        'log_alpha': np.log(alpha),
        'test_mse': -np.mean(ridge_cv['test_score'])
    })

# Convert the cross-validation scores into a DataFrame
ridge_cv_scores_df = pd.DataFrame(ridge_cv_scores)
print(ridge_cv_scores_df)

# Plot the cross-validation scores as a function of alpha
fig = px.line(ridge_cv_scores_df,
              x='log_alpha',
              y='test_mse',
              title='Ridge Regression Cross-Validation Scores',
              labels={'log_alpha': 'Log(Alpha)', 'test_mse': 'Test RMSE'})

fig.show()

```

	alpha	log_alpha	test_mse
0	0.100000	-2.302585	1311.335987
1	0.117681	-2.139776	1311.335987
2	0.138489	-1.976967	1311.335987
3	0.162975	-1.814158	1311.335987
4	0.191791	-1.651349	1311.335987
5	0.225702	-1.488540	1311.335987
6	0.265609	-1.325731	1311.335987
7	0.312572	-1.162922	1311.335987
8	0.367838	-1.000113	1311.335987
9	0.432876	-0.837304	1311.335987
10	0.509414	-0.674495	1311.335987
11	0.599484	-0.511686	1311.335987
12	0.705480	-0.348877	1311.335987
13	0.830218	-0.186067	1311.335987
14	0.977010	-0.023258	1311.335987
15	1.149757	0.139551	1311.335987
16	1.353048	0.302360	1311.335987
17	1.592283	0.465169	1311.335987
18	1.873817	0.627978	1311.335987
19	2.205131	0.790787	1311.335987
20	2.595024	0.953596	1311.335987
21	3.053856	1.116405	1311.335987
22	3.593814	1.279214	1311.335987
23	4.229243	1.442023	1311.335987
24	4.977024	1.604832	1311.335987
25	5.857021	1.767641	1311.335987
26	6.892612	1.930450	1311.335987
27	8.111308	2.093259	1311.335987
28	9.545485	2.256068	1311.335987
29	11.233240	2.418877	1311.335987
30	13.219411	2.581686	1311.335987
31	15.556761	2.744495	1311.335987
32	18.307383	2.907304	1311.335987
33	21.544347	3.070113	1311.335987
34	25.353645	3.232923	1311.335988
35	29.836472	3.395732	1311.335988
36	35.111917	3.558541	1311.335988
37	41.320124	3.721350	1311.335988
38	48.626016	3.884159	1311.335988
39	57.223677	4.046968	1311.335988
40	67.341507	4.209777	1311.335988
41	79.248290	4.372586	1311.335988
42	93.260335	4.535395	1311.335989
43	109.749877	4.698204	1311.335989
44	129.154967	4.861013	1311.335990
45	151.991108	5.023822	1311.335991
46	178.864953	5.186631	1311.335993
47	210.490414	5.349440	1311.335994
48	247.707636	5.512249	1311.335997
49	291.505306	5.675058	1311.336001
50	343.046929	5.837867	1311.336006
51	403.701726	6.000676	1311.336013
52	475.081016	6.163485	1311.336022
53	559.081018	6.326294	1311.336036
54	657.933225	6.489103	1311.336054
55	774.263683	6.651912	1311.336079
56	911.162756	6.814722	1311.336114
57	1072.267222	6.977531	1311.336162
58	1261.856883	7.140340	1311.336229
59	1484.968262	7.303149	1311.336320
60	1747.528400	7.465958	1311.336447
61	2056.512308	7.628767	1311.336622
62	2420.128265	7.791576	1311.336864
63	2848.035868	7.954385	1311.337197
64	3351.602651	8.117194	1311.337655
65	3944.206059	8.280003	1311.338287
66	4641.588834	8.442812	1311.339157
67	5462.277218	8.605621	1311.340352
68	6428.073117	8.768430	1311.341993
69	7564.633276	8.931239	1311.344243
70	8902.150854	9.094048	1311.347323
71	10476.157528	9.256857	1311.351531
72	12328.467394	9.419666	1311.357265
73	14508.287785	9.582475	1311.365059
74	17073.526475	9.745284	1311.375620
75	20092.330026	9.908093	1311.389879
76	23644.894126	10.070902	1311.409049
77	27825.594022	10.233712	1311.434698
78	32745.491629	10.396521	1311.468823
79	38535.285937	10.559330	1311.513938
80	45348.785081	10.722139	1311.573147
81	53366.992312	10.884948	1311.650212
82	62802.914418	11.047757	1311.749584
83	73907.220335	11.210566	1311.876382

83	73907.220335	11.210566	1311.876382
84	86974.900262	11.373375	1312.036295
85	102353.102190	11.536184	1312.235380
86	120450.354026	11.698993	1312.479748
87	141747.416293	11.861802	1312.775119
88	166810.053720	12.024611	1313.126287
89	196304.065004	12.187420	1313.536525
90	231012.970008	12.350229	1314.007006
91	271858.824273	12.513038	1314.536347
92	319926.713780	12.675847	1315.120344
93	376493.580679	12.838656	1315.751986
94	443062.145758	13.001465	1316.421771
95	521400.828800	13.164274	1317.118287
96	613590.727341	13.327083	1317.828998
97	722080.901839	13.489892	1318.541113
98	849753.435909	13.652702	1319.242419
99	1000000.000000	13.815511	1319.921986



```
# @title
# First, split the data into training and temp (which will later be split into validation and testing)
train_data, temp_data = train_test_split(df, test_size=0.4, random_state=42) # the random state will ensure we will all start

# Now split the temp data into validation and testing sets
val_data, test_data = train_test_split(temp_data, test_size=0.5, random_state=42)

# Check the sizes of each set
print(f'Training set size: {train_data.shape[0]}')
print(f'Validation set size: {val_data.shape[0]}')
print(f'Test set size: {test_data.shape[0]}')

train_data.sample(10)

train_data.sample(1)
```

```
Training set size: 348607
Validation set size: 116202
Test set size: 116203
```

	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_Roa
409108	3270	32	10	361	66	

```
# @title
# Select only the necessary features for logistic regression
features = ['Elevation', 'Aspect', 'Slope', 'Horizontal_Distance_To_Hydrology',
       'Vertical_Distance_To_Hydrology', 'Horizontal_Distance_To_Roadways',
       'Hillshade_9am', 'Hillshade_Noon', 'Hillshade_3pm',
       'Horizontal_Distance_To_Fire_Points', 'Wilderness_Area1', 'Wilderness_Area2', 'Wilderness_Area3',
       'Wilderness_Area4']
lr_df = train_data[['Target'] + features]

X_train = lr_df[features]
X_val = val_data[features]
y_train = train_data['Target']
y_val = val_data['Target']

lr = LogisticRegression(solver = 'liblinear')
lr.fit(X = X_train, y = y_train)

lr_intercept = lr.intercept_
lr_coefficients = lr.coef_

print(lr_intercept, lr_coefficients)
```

```
[-0.00788368  0.03675706  0.08635204  0.04193273 -0.04632042 -0.02042593
 -0.09622012] [[ 6.15048992e-03 -4.21220501e-04 -8.78639608e-02 -1.57024112e-03
  -2.63569108e-03 -6.39209791e-05 -7.58761553e-02  2.93971943e-02
  -5.26363405e-02 -2.80861056e-05  1.05002981e-01 -3.64033477e-02
  -5.93899565e-02 -1.70933525e-02]
 [-4.32453017e-03 -1.57159978e-04  4.10523706e-02  1.64245851e-03
   5.00947753e-04  5.29863572e-05  4.65588529e-02 -2.13999842e-02
   3.71661630e-02  9.40015259e-05  1.43507050e+00  1.24845455e+00
   1.08815040e+00 -3.73491840e+00]
 [-8.80924280e-03  1.21444853e-03  1.13823722e-01  2.34287993e-03
   2.86433179e-03 -1.44225928e-04  7.54751247e-02 -2.09957045e-02
   4.84097857e-02 -2.20612882e-04 -1.60297431e+00 -2.20895223e-02
   1.21791486e+00  4.93501010e-01]
 [-6.93009151e-03  7.18830941e-04 -4.73748565e-02 -8.61774595e-03
   1.35676769e-02  8.54366715e-04  2.01687718e-02  3.87310122e-02
  -1.65686764e-02  3.55298938e-04 -1.59708981e+00 -4.06333372e-03
  -1.38251701e+00  3.02560289e+00]
 [-2.67592798e-03  2.78149383e-03  1.06815713e-02 -1.63756090e-03
   6.32372357e-03 -4.84227117e-04  7.36556049e-03  1.32083326e-02
  -1.29243300e-02 -1.02882424e-04  1.62187127e+00 -5.76868852e-01
   1.56820392e+00 -2.65952675e+00]
 [-6.95318196e-03 -6.59951869e-05  2.86176689e-02 -1.45074377e-03
   1.01335941e-03  6.81080378e-05  1.37330317e-01 -1.48931879e-01
   1.28815056e-01  7.27498577e-05 -1.46643570e+00 -4.16216972e-02
   1.43273430e+00  5.48971694e-02]
 [ 1.46888252e-02  4.91315247e-04 -2.62495755e-01 -1.54648938e-03
  -1.00422117e-03  6.16944844e-05 -2.64901397e-01  1.83531667e-01
  -2.21973550e-01  1.89746426e-04 -7.09264192e-01 -3.77331519e-01
   9.93062789e-01 -2.68719664e-03]]
```

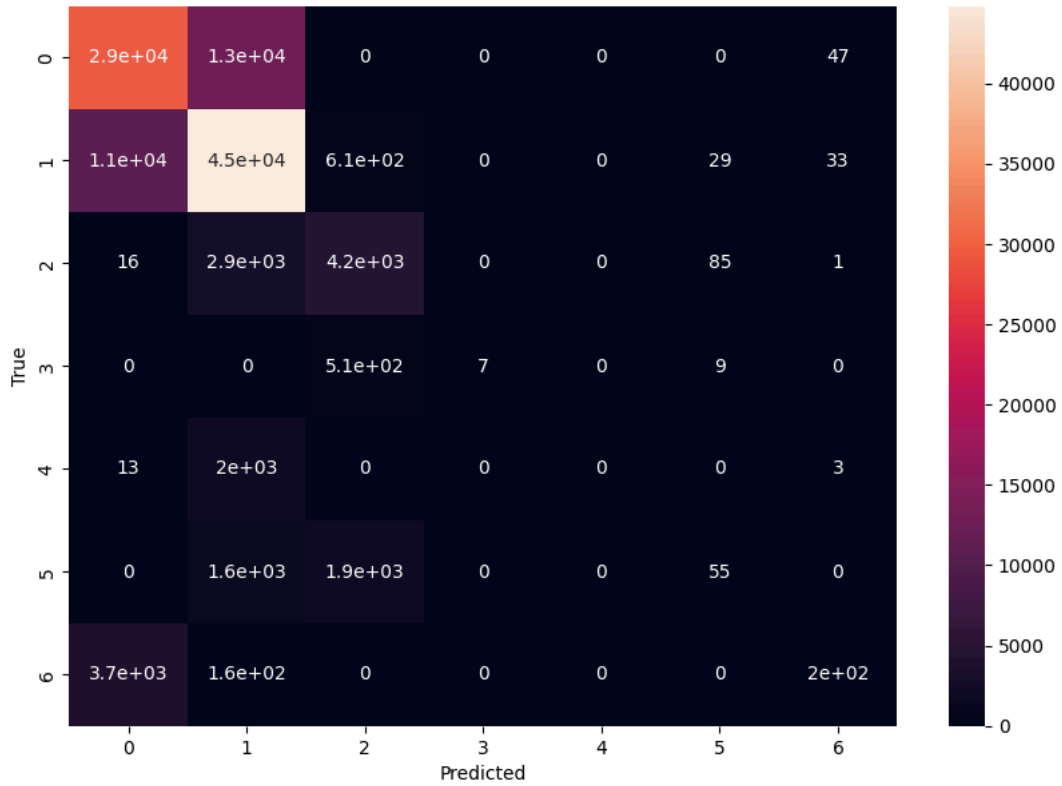
```
# @title
pred_val = pd.DataFrame(dict(
    target = y_val,
    lr_predict = lr.predict(X_val)))
print(lr.score(X_val, y_val))
pred_val.sample(10)
```

```
0.676709523071892
```

	target	lr_predict
206959	1	1
326038	5	2
169403	1	1
140949	2	2
498118	1	2
234189	2	2
153292	2	2
497401	7	1
554295	2	2
206502	1	1

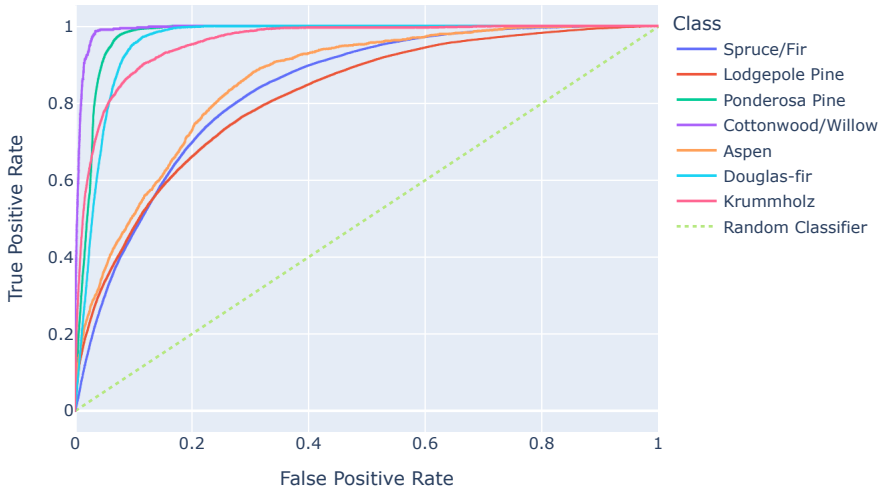
```
# @title
# Confusion matrix
conf_lr = metrics.confusion_matrix(y_true = pred_val['target'],
                                   y_pred = pred_val['lr_predict'])

import seaborn as sns
plt.figure(figsize = (10, 7))
sns.heatmap(conf_lr, annot = True)
plt.xlabel('Predicted')
plt.ylabel('True')
```

```
Text(95.72222222222221, 0.5, 'True')
```




```
# @title
from sklearn.preprocessing import label_binarize
from sklearn.metrics import roc_curve, auc
import plotly.graph_objs as go
```

```

false_pos_rate = dict()
true_pos_rate = dict()
auc = dict()
roc_data = []

cover_type_map = {
    1: 'Spruce/Fir', 2: 'Lodgepole Pine', 3: 'Ponderosa Pine',
    4: 'Cottonwood/Willow', 5: 'Aspen', 6: 'Douglas-fir', 7: 'Krummholz'}

# binarize target variable
classes = np.unique(pred_val['target'])
y_true = label_binarize(pred_val['target'], classes=classes)

# get prediction probabilities
y_pred_proba = lr.predict_proba(X_val)

# compute ROC and AUC for each class
for i in range(len(classes)):
    false_pos_rate[i], true_pos_rate[i], _ = roc_curve(y_true[:,i], y_pred_proba[:,i])
    auc[i] = metrics.auc(false_pos_rate[i], true_pos_rate[i])

for i in range(len(classes)):
    class_roc = pd.DataFrame({
        'False Positive Rate': false_pos_rate[i],
        'True Positive Rate': true_pos_rate[i],
        'Class': cover_type_map[classes[i]],
        'Threshold': np.linspace(0, 1, len(false_pos_rate[i]))})
    roc_data.append(class_roc)

# plot ROCs
roc_df = pd.concat(roc_data)
fig = px.line(roc_df, x='False Positive Rate', y='True Positive Rate',
              color='Class', width=700, height=500,
              title='ROC Curves for Each Forest Cover Type')
fig.add_trace(go.Scatter(x=[0, 1], y=[0, 1],
                         mode='lines',
                         name='Random Classifier',
                         line=dict(dash='dot')))
fig.show()

# print AUC for each cover type, as well as average AUC
print("\nAUC for each forest cover type:")
for i, auc_score in enumerate(auc.values()):
    print(f"{cover_type_map[classes[i]]}: {auc_score:.4f}")
average_auc = np.mean(list(auc.values()))
print(f"\nAverage AUC: {average_auc:.4f}")
```

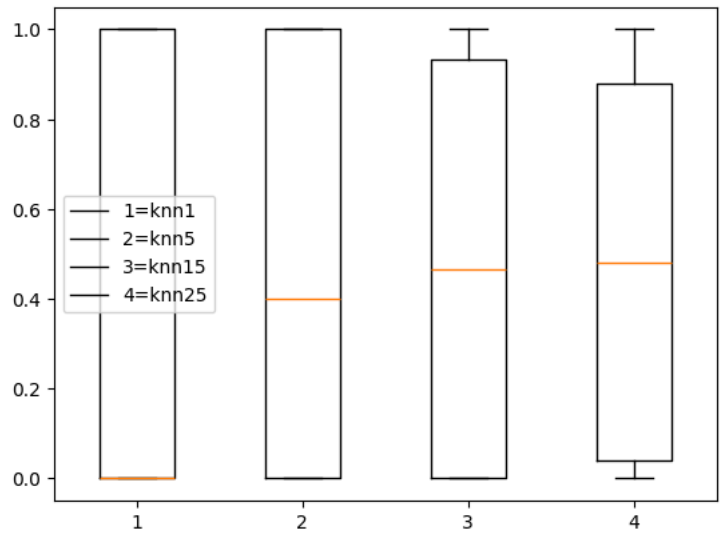



```
AUC for each forest cover type:
Spruce/Fir: 0.8340
Lodgepole Pine: 0.8141
Ponderosa Pine: 0.9760
Cottonwood/Willow: 0.9928
Aspen: 0.8554
Douglas-fir: 0.9620
Krummholz: 0.9580

Average AUC: 0.9132
```

Implementing KNN model using best predictor variables chosen from PCA (['Horizontal_Distance_To_Fire_Points'], ['Horizontal_Distance_To_Roadways']) with various k values to select one for best accuracy.

```
# @title

# Correct way to select multiple columns
X = train_data[['Elevation', 'Aspect', 'Slope', 'Horizontal_Distance_To_Hydrology',
       'Vertical_Distance_To_Hydrology', 'Horizontal_Distance_To_Roadways',
       'Hillshade_9am', 'Hillshade_Noon', 'Hillshade_3pm',
       'Horizontal_Distance_To_Fire_Points', 'Wilderness_Area1', 'Wilderness_Area2', 'Wilderness_Area3',
       'Wilderness_Area4']]
y = train_data['Target']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

knn1 = KNeighborsClassifier(1)
knn1.fit(X_train_scaled, y_train)

knn5 = KNeighborsClassifier(5)
knn5.fit(X_train_scaled, y_train)

knn15 = KNeighborsClassifier(15)
knn15.fit(X_train_scaled, y_train)

knn25 = KNeighborsClassifier(25)
knn25.fit(X_train_scaled, y_train)

pred1 = knn1.predict(X_train_scaled)
pred5 = knn5.predict(X_train_scaled)
pred15 = knn15.predict(X_train_scaled)
pred25 = knn25.predict(X_train_scaled)

plt.boxplot([knn1.predict_proba(X_train_scaled)[:,1],knn5.predict_proba(X_train_scaled)[:,1],
```

```
            knn15.predict_proba(X_train_scaled)[:,1],knn25.predict_proba(X_train_scaled)[:,1]])
plt.legend(["1=knn1","2=knn5","3=knn15","4=knn25"]);
```

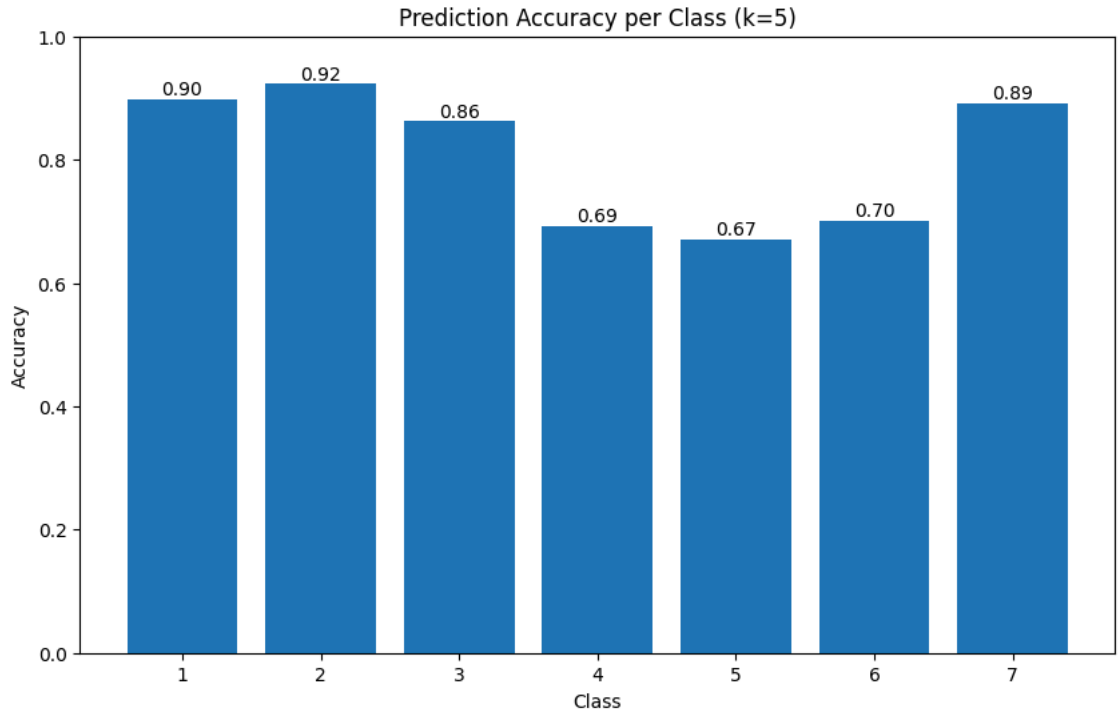



```
# @title
print("Classification accuracy for knn1 were: \n Train =",
      knn1.score(X_train_scaled, y_train), ", Test =",
      knn1.score(X_test_scaled, y_test))

print("Classification accuracy for knn5 were: \n Train =",
      knn5.score(X_train_scaled, y_train), ", Test =",
      knn5.score(X_test_scaled, y_test))

print("Classification accuracy for knn15 were: \n Train =",
      knn15.score(X_train_scaled, y_train), ", Test =",
      knn15.score(X_test_scaled, y_test))

print("Classification accuracy for knn25 were: \n Train =",
      knn25.score(X_train_scaled, y_train), ", Test =",
      knn25.score(X_test_scaled, y_test))
```

```
Classification accuracy for knn1 were:
 Train = 1.0 , Test = 0.907618829063997
Classification accuracy for knn5 were:
 Train = 0.9372680495544758 , Test = 0.8976793551533232
Classification accuracy for knn15 were:
 Train = 0.8939132617387096 , Test = 0.8740426264306819
Classification accuracy for knn25 were:
 Train = 0.8728077881564086 , Test = 0.8579214595106279
```

```
# @title


y_pred = knn5.predict(X_test_scaled)

# accuracy for each class
accuracies = []
for class_num in range(1, 8):  # for classes 1-7
    # Get indices where true value is the current class
    class_indices = y_test == class_num
    # Calculate accuracy for this class
    class_accuracy = np.mean(y_pred[class_indices] == y_test[class_indices])
    accuracies.append(class_accuracy)

# Create bar plot
plt.figure(figsize=(10, 6))
plt.bar(range(1, 8), accuracies)
plt.title('Prediction Accuracy per Class (k=5)')
plt.xlabel('Class')
plt.ylabel('Accuracy')
plt.ylim(0, 1)  # Set y-axis from 0 to 1
plt.xticks(range(1, 8))
for i, acc in enumerate(accuracies):
    plt.text(i+1, acc, f'{acc:.2f}', ha='center', va='bottom')
plt.show()
```

```
26
27 # Print numeric values
28 for i, acc in enumerate(accuracies, 1):
29     print(f"Class {i} accuracy: {acc:.3f}")
30
```




```
Class 1 accuracy: 0.898
Class 2 accuracy: 0.923
Class 3 accuracy: 0.863
Class 4 accuracy: 0.693
Class 5 accuracy: 0.671
Class 6 accuracy: 0.701
Class 7 accuracy: 0.892
```

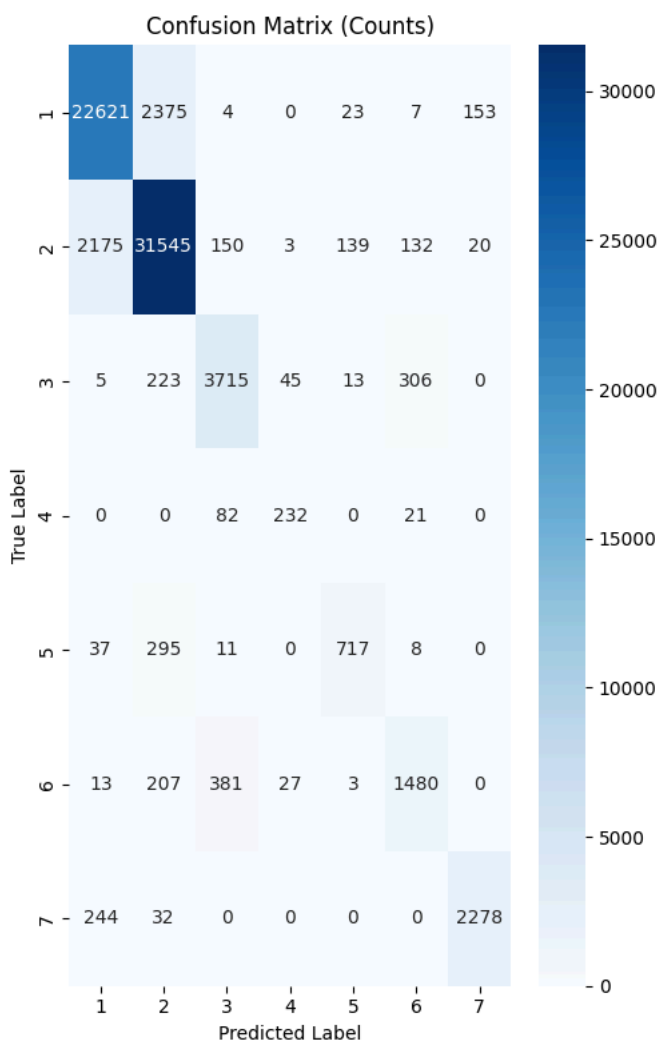
```
1 # @title
2
3 # make predictions on test data
4 # choose k = 5, bc best balance of accuracy and not too different in overfitting data like k=1
5 test_pred = y_pred
6 test_pred_proba = knn5.predict_proba(X_test_scaled)
7
8 # Confusion Matrix on test data
9 conf_matrix = confusion_matrix(y_test, test_pred)
10 print("\nConfusion Matrix (Test Data):")
11 print(conf_matrix)
```

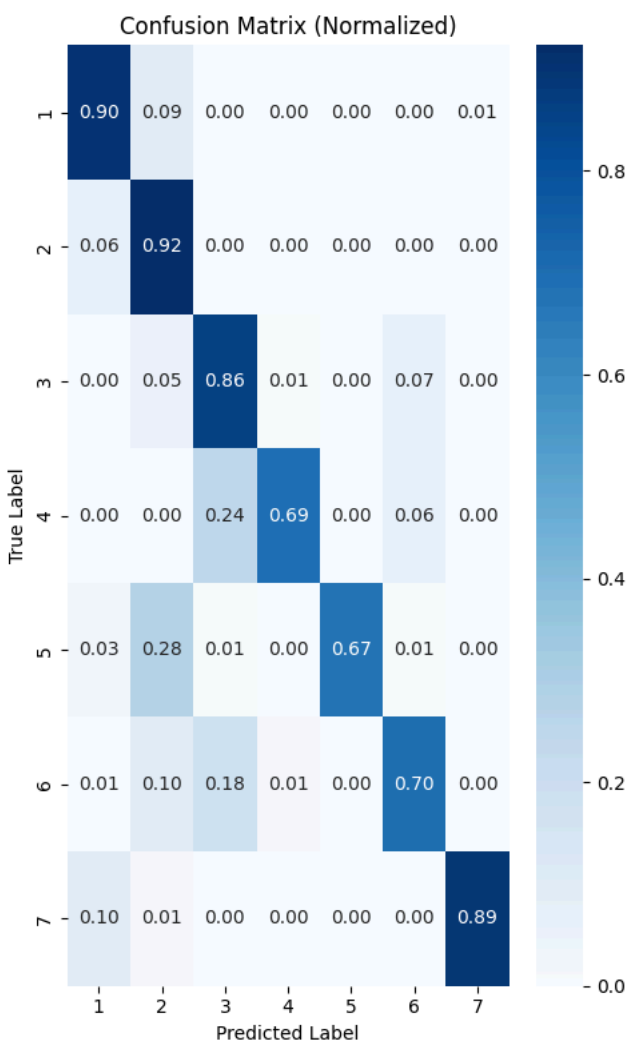
```
Confusion Matrix (Test Data):
[[22621  2375     4     0    23     7   153]
 [ 2175 31545   150     3   139   132    20]
 [    5   223  3715    45    13   306     0]
 [    0     0    82   232     0    21     0]
 [   37   295    11     0   717     8     0]
 [   13   207   381    27     3  1480     0]
 [  244    32     0     0     0     0  2278]]
```

```
1 # @title
2 # metrics for entire model and per class confusion matrix
3 accuracy = accuracy_score(y_test, test_pred)
4 error_rate = 1 - accuracy
5
6 f1 = f1_score(y_test, test_pred, average='weighted')
7
8 print(f"\nTest Metrics:")
9 print(f"Accuracy: {accuracy:.3f}")
10 print(f"Error Rate: {error_rate:.3f}")
11 print("\nPer-class True Negative Rates:")
12 print(f"\nF1 Score (weighted): {f1:.3f}")
```

```
Test Metrics:
Accuracy: 0.898
Error Rate: 0.102

Per-class True Negative Rates:

F1 Score (weighted): 0.897
```

```
# @title

# assembling confusion matrix per class with TP, TN, FP, FN rates

conf_matrix = confusion_matrix(y_test, y_pred)


labels = range(1, 8)  # for classes 1-7

plt.figure(figsize=(10, 8))

# raw counts of prediction
plt.subplot(1, 2, 1)
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=labels, yticklabels=labels)
plt.title('Confusion Matrix (Counts)')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')

# plot percentages by normalizing
plt.subplot(1, 2, 2)
conf_matrix_norm = conf_matrix.astype('float') / conf_matrix.sum(axis=1)[:, np.newaxis]
sns.heatmap(conf_matrix_norm, annot=True, fmt='.2f', cmap='Blues',
            xticklabels=labels, yticklabels=labels)
plt.title('Confusion Matrix (Normalized)')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')

plt.tight_layout()
plt.show()


print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```






```
Classification Report:
              precision    recall  f1-score   support

           1       0.90      0.90      0.90     25183
           2       0.91      0.92      0.92     34164
           3       0.86      0.86      0.86      4307
           4       0.76      0.69      0.72       335
           5       0.80      0.67      0.73      1068
           6       0.76      0.70      0.73      2111
           7       0.93      0.89      0.91      2554

    accuracy                           0.90     69722
   macro avg       0.84      0.81      0.82     69722
weighted avg       0.90      0.90      0.90     69722
```

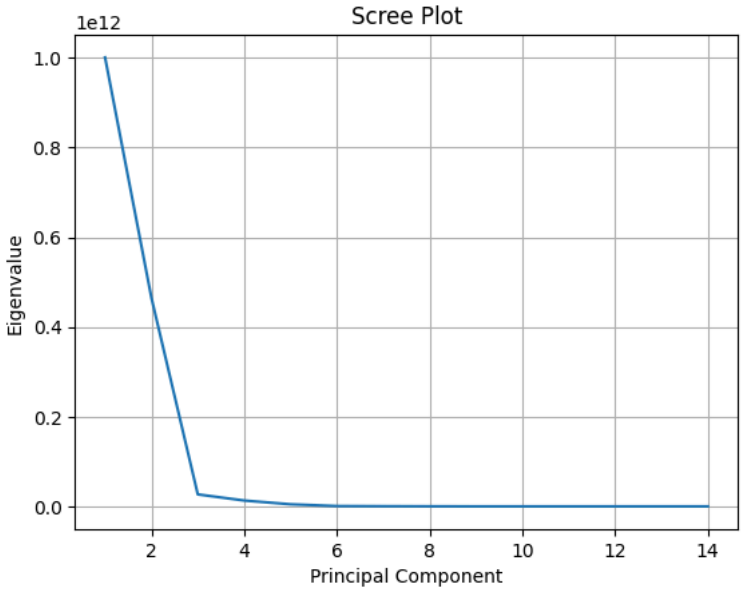
Checkin 5: PCA

```
# @title
# Convert to numpy array
count_mat = X.to_numpy()

# Mean-center the columns
X = count_mat - np.mean(count_mat, axis=0)

# Calculate covariance matrix (X^T X)
cov_mat = np.matmul(np.transpose(X), X)

# Calculate eigenvalues and eigenvectors
eigenvals, eigenvecs = np.linalg.eig(cov_mat)

# Print first 3 eigenvalues
print("First eigenvals:", eigenvals[0:3])
print("\nFirst eigenvecs:", eigenvecs[:, 0:3])
```

```
First eigenvals: [1.00116106e+12 4.65113084e+11 2.66989388e+10]

First eigenvecs: [[ 5.92247672e-02 -2.96008310e-02  8.75886073e-01]
 [-1.70603493e-03 -1.21937889e-02  8.47228985e-03]
 [-1.09245248e-03 -1.68915441e-04 -3.62472966e-03]
 [ 1.02349449e-02 -7.13825377e-04  4.72897856e-01]
 [-2.24616624e-03 -1.83746588e-03  7.08367335e-02]
 [ 8.48630976e-01 -5.24739068e-01 -6.22121735e-02]
 [ 1.30794569e-03  2.42340874e-03  6.52530375e-03]
 [ 2.02904012e-03 -1.38683345e-03  9.53408442e-03]
 [ 1.43580788e-03 -4.08587067e-03  5.68618933e-03]
 [ 5.25543594e-01  8.50644229e-01 -7.20292687e-03]
 [ 1.50665592e-04  2.07180933e-05 -1.70067498e-04]
 [-1.86394287e-05  3.20426513e-05  2.40385040e-04]
 [-8.64399064e-05 -4.53336951e-05  3.56534427e-04]
 [-4.55862570e-05 -7.42704946e-06 -4.26851969e-04]]
```

```python
# @title
# plotting a scree plot
# Check how many eigenvalues you actually have
n_components = len(sorted_eigenvals)
print(f"Number of components: {n_components}")

# Plot using the actual number of components
plt.plot(range(1, n_components + 1), sorted_eigenvals)
plt.title('Scree Plot')
plt.xlabel('Principal Component')
plt.ylabel('Eigenvalue')
plt.grid(True)
plt.show()
```

```
Number of components: 14
```



Having a larger first eigenvector compared to other eigenvectors, and the steep drop off in the scree plot after the first few components indicates to us that we can indeed just use a few components to determine covertype than all of the original features.

```python
# @title
# Print cumulative variance explained
cumulative_variance = np.cumsum(pca.explained_variance_ratio_)
print("\nCumulative variance explained by components:")
for i, var in enumerate(cumulative_variance):
    if var > 0.9:  # Stop after we reach 90% explained variance
        print(f"First {i+1} components explain {var:.2%} of variance")
        break
    print(f"First {i+1} components explain {var:.2%} of variance")
```

```
Cumulative variance explained by components:
First 1 components explain 66.22% of variance
First 2 components explain 96.99% of variance
```

This means we just need 2 components to capture about 97% of the data. Now we will run pca with the goal of just using the top 2 components.

```

# Get feature names from the variables DataFrame
feature_names = covertype.variables['name'].tolist()

# Create and fit PCA
pca = PCA(n_components=2, svd_solver='full')
pca.fit(X)

# Print explained variance
print("\nCumulative variance explained by components:")
cumulative_variance = np.cumsum(pca.explained_variance_ratio_)
for i, var in enumerate(cumulative_variance):
    print(f"First {i+1} components explain {var:.2%} of variance")

# Print top contributing features for each component
for i, component in enumerate(pca.components_):
    # Get indices of top 5 features by absolute value
    top_features = np.argsort(np.abs(component))[-5:][::-1]

    print(f"\nTop 5 features in component {i+1}:")
    for idx in top_features:
        print(f"{feature_names[idx]}: {component[idx]:.4f}")
```

```
Cumulative variance explained by components:
First 1 components explain 66.22% of variance
First 2 components explain 96.99% of variance

Top 5 features in component 1:
Horizontal_Distance_To_Roadways: 0.8486
Horizontal_Distance_To_Fire_Points: 0.5255
Elevation: 0.0592
Horizontal_Distance_To_Hydrology: 0.0102
Vertical_Distance_To_Hydrology: -0.0022

Top 5 features in component 2:
Horizontal_Distance_To_Fire_Points: 0.8506
Horizontal_Distance_To_Roadways: -0.5247
Elevation: -0.0296
Aspect: -0.0122
Hillshade_3pm: -0.0041
```

```
# @title

```

Checkin 6: Neural Networks

```
from sklearn.preprocessing import StandardScaler, LabelEncoder
```

```
## Prep Work ##
# data (as pandas dataframes)
X = covertype.data.features
X = X.drop(columns=X.filter(regex='^(Soil_Type|Wilderness_Area)').columns)
y = covertype.data.targets

print(X.shape)
print(y.shape)

# First, split the data into training and temp (which will later be split into validation and testing
X_temp, X_test, y_temp, y_test = train_test_split(
    X, y, test_size=10000, random_state=58, stratify=y)

X_train, X_valid, y_train, y_valid = train_test_split(
    X_temp, y_temp, test_size=5000, random_state=58, stratify=y_temp)

# Check the sizes of each set
print(f'Training set size: {X_train.shape[0]}')
print(f'Validation set size: {X_valid.shape[0]}')
print(f'Test set size: {X_test.shape[0]}')
```

Checkin 6: Neural Networks

```
from sklearn.preprocessing import StandardScaler, LabelEncoder
```

```
## Prep Work ##
# data (as pandas dataframes)
X = covertype.data.features
X = X.drop(columns=X.filter(regex='^(Soil_Type|Wilderness_Area)').columns)
y = covertype.data.targets

print(X.shape)
print(y.shape)

# First, split the data into training and temp (which will later be split into validation and testing)
X_temp, X_test, y_temp, y_test = train_test_split(
    X, y, test_size=10000, random_state=58, stratify=y)

X_train, X_valid, y_train, y_valid = train_test_split(
    X_temp, y_temp, test_size=5000, random_state=58, stratify=y_temp)

# Check the sizes of each set
print(f'Training set size: {X_train.shape[0]}')
print(f'Validation set size: {X_valid.shape[0]}')
print(f'Test set size: {X_test.shape[0]}')

# Check the sizes of each set
print(f'Training set size: {y_train.shape[0]}')
print(f'Validation set size: {y_valid.shape[0]}')
print(f'Test set size: {y_test.shape[0]}')

# Normalize features using StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_valid_scaled = scaler.transform(X_valid)
X_test_scaled = scaler.transform(X_test)

# Label Encoding to ensure target labels are integers
label_encoder = LabelEncoder()

# Convert y_train and y_valid into integer class labels (if they are not already integers)
y_train = label_encoder.fit_transform(y_train)
y_valid = label_encoder.transform(y_valid)
```

```
(581012, 10)
(581012, 1)
Training set size: 566012
Validation set size: 5000
Test set size: 10000
Training set size: 566012
Validation set size: 5000
Test set size: 10000
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_label.py:114: DataConversionWarning:

A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using

/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_label.py:132: DataConversionWarning:

A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using
```

```
# Define sigmoid function
def sigmoid(z):
    return 1. / (1. + np.exp(-z))

# Assigns hot encoding to the Target variable (e.g. if Target = 2 then [0, 1, 0, 0, 0, 0])
def int_to_onehot(y, num_labels):
    ary = np.zeros((y.shape[0], num_labels))
    for i, val in enumerate(y):
        ary[i, val] = 1
    return ary

# Define softmax function for multi-class classification
def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))  # For numerical stability
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)

```

```
## Implement the NN ##
# Neural Network Class
class NeuralNetMLP:

    def __init__(self, num_features, num_hidden, num_classes, random_seed=123):
        super().__init__()

        self.num_classes = num_classes

        # Initialize weights and biases using random normal distribution
        rng = np.random.RandomState(random_seed)

        self.weight_h = rng.normal(loc=0.0, scale=0.1, size=(num_hidden, num_features))
        self.bias_h = np.zeros(num_hidden)

        self.weight_out = rng.normal(loc=0.0, scale=0.1, size=(num_classes, num_hidden))
        self.bias_out = np.zeros(num_classes)

    def forward(self, x):
        # Hidden layer
        z_h = np.dot(x, self.weight_h.T) + self.bias_h
        a_h = sigmoid(z_h)

        # Output layer (Softmax for multi-class classification)
        z_out = np.dot(a_h, self.weight_out.T) + self.bias_out
        a_out = softmax(z_out)
        return a_h, a_out

    def backward(self, x, a_h, a_out, y):
        y_onehot = int_to_onehot(y, self.num_classes)

        # Output layer gradient
        d_loss__d_a_out = a_out - y_onehot
        d_a_out__d_z_out = a_out * (1. - a_out)
        delta_out = d_loss__d_a_out * d_a_out__d_z_out

        d_z_out__dw_out = a_h
        d_loss__dw_out = np.dot(delta_out.T, d_z_out__dw_out)
        d_loss__db_out = np.sum(delta_out, axis=0)

        # Hidden layer gradient
        d_z_out__a_h = self.weight_out
        d_loss__a_h = np.dot(delta_out, d_z_out__a_h)
        d_a_h__d_z_h = a_h * (1. - a_h)
        d_z_h__d_w_h = x
        d_loss__d_w_h = np.dot((d_loss__a_h * d_a_h__d_z_h).T, d_z_h__d_w_h)
        d_loss__d_b_h = np.sum(d_loss__a_h * d_a_h__d_z_h, axis=0)

        return d_loss__dw_out, d_loss__db_out, d_loss__d_w_h, d_loss__d_b_h

    def compute_loss(self, y_true, y_pred):
        m = y_true.shape[0]
        log_likelihood = -np.log(y_pred[range(m), np.argmax(y_true, axis=1)])
        loss = np.sum(log_likelihood) / m
        return loss

    def compute_accuracy(self, y_true, y_pred):
        y_true_class = np.argmax(y_true, axis=1)
        return np.mean(y_true_class == y_pred)

    def predict(self, X):
        _, a_out = self.forward(X)
        return np.argmax(a_out, axis=1)
```

```
## Initialize the model ##
model = NeuralNetMLP(X_train_scaled.shape[1],
                     num_hidden=96,
                     num_classes=7)
```

```
# Data Loaders
num_epochs = 50
minibatch_size = 32

## Generates "mini-batches" of the training data to train the model on random subets of the train data
def minibatch_generator(X, y, minibatch_size):
    indices = np.arange(X.shape[0])
```

```
    np.random.shuffle(indices)

    for start_idx in range(0, indices.shape[0] - minibatch_size + 1, minibatch_size):
        batch_idx = indices[start_idx:start_idx + minibatch_size]
        yield X[batch_idx], y[batch_idx]

# Training loop
def train(model, X_train, y_train, X_valid, y_valid, epochs=50, batch_size=100, learning_rate=0.01):
    losses = []
    val_losses = []
    accuracies = []  # Store training accuracy per epoch
    val_accuracies = []  # Store validation accuracy per epoch

    # One-hot encode labels
    y_train_onehot = int_to_onehot(y_train, model.num_classes)
    y_valid_onehot = int_to_onehot(y_valid, model.num_classes)

    for epoch in range(epochs):
        minibatch_gen = minibatch_generator(X_train, y_train, batch_size)

        for X_batch, y_batch in minibatch_gen:
            # Forward pass
            a_h, a_out = model.forward(X_batch)

            # Backward pass
            d_dw_out, d_db_out, d_d_w_h, d_d_b_h = model.backward(X_batch, a_h, a_out, y_batch)

            # Update weights and biases using grad descent
            model.weight_out -= learning_rate * d_dw_out
            model.bias_out -= learning_rate * d_db_out
            model.weight_h -= learning_rate * d_d_w_h
            model.bias_h -= learning_rate * d_d_b_h

        # Compute loss and accuracy after each epoch
        train_loss = model.compute_loss(y_train_onehot, model.forward(X_train)[1])
        train_acc = model.compute_accuracy(y_train_onehot, model.predict(X_train))

        val_loss = model.compute_loss(y_valid_onehot, model.forward(X_valid)[1])
        val_acc = model.compute_accuracy(y_valid_onehot, model.predict(X_valid))

        losses.append(train_loss)
        val_losses.append(val_loss)
        accuracies.append(train_acc)
        val_accuracies.append(val_acc)

        print(f"Epoch {epoch+1}/{epochs}, Train Loss: {train_loss:.4f}, Train Accuracy: {train_acc*100:.2f}%")
        print(f"Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_acc*100:.2f}%")

    return losses, val_losses, accuracies, val_accuracies


# Train the model
losses, val_losses, accuracies, val_accuracies = train(model, X_train_scaled, y_train, X_valid_scaled, y_valid, epochs=50, ba
```

```
Epoch 1/50, Train Loss: 0.8591, Train Accuracy: 69.41%
Validation Loss: 0.8701, Validation Accuracy: 68.86%
Epoch 2/50, Train Loss: 0.8530, Train Accuracy: 69.56%
Validation Loss: 0.8902, Validation Accuracy: 69.46%
Epoch 3/50, Train Loss: 0.7375, Train Accuracy: 71.86%
Validation Loss: 0.7659, Validation Accuracy: 71.50%
Epoch 4/50, Train Loss: 0.7333, Train Accuracy: 71.27%
Validation Loss: 0.7607, Validation Accuracy: 71.08%
Epoch 5/50, Train Loss: 0.7097, Train Accuracy: 72.48%
Validation Loss: 0.7423, Validation Accuracy: 72.18%
Epoch 6/50, Train Loss: 0.6873, Train Accuracy: 73.71%
Validation Loss: 0.7240, Validation Accuracy: 73.00%
Epoch 7/50, Train Loss: 0.6705, Train Accuracy: 74.10%
Validation Loss: 0.6970, Validation Accuracy: 73.42%
Epoch 8/50, Train Loss: 0.6545, Train Accuracy: 74.16%
Validation Loss: 0.6728, Validation Accuracy: 73.58%
Epoch 9/50, Train Loss: 0.6461, Train Accuracy: 74.69%
Validation Loss: 0.6576, Validation Accuracy: 74.12%
Epoch 10/50, Train Loss: 0.6437, Train Accuracy: 74.87%
Validation Loss: 0.6494, Validation Accuracy: 74.48%
Epoch 11/50, Train Loss: 0.6403, Train Accuracy: 75.11%
Validation Loss: 0.6433, Validation Accuracy: 74.62%
Epoch 12/50, Train Loss: 0.6348, Train Accuracy: 75.24%
Validation Loss: 0.6396, Validation Accuracy: 74.88%
Epoch 13/50, Train Loss: 0.6256, Train Accuracy: 75.39%
```

```
Validation Loss: 0.6318, Validation Accuracy: 74.92%
Epoch 14/50, Train Loss: 0.6204, Train Accuracy: 75.54%
Validation Loss: 0.6293, Validation Accuracy: 74.76%
Epoch 15/50, Train Loss: 0.6161, Train Accuracy: 75.77%
Validation Loss: 0.6240, Validation Accuracy: 75.36%
Epoch 16/50, Train Loss: 0.6170, Train Accuracy: 75.97%
Validation Loss: 0.6264, Validation Accuracy: 75.16%
Epoch 17/50, Train Loss: 0.6209, Train Accuracy: 76.17%
Validation Loss: 0.6321, Validation Accuracy: 75.40%
Epoch 18/50, Train Loss: 0.6226, Train Accuracy: 76.20%
Validation Loss: 0.6359, Validation Accuracy: 75.68%
Epoch 19/50, Train Loss: 0.6277, Train Accuracy: 76.18%
Validation Loss: 0.6416, Validation Accuracy: 75.68%
Epoch 20/50, Train Loss: 0.6326, Train Accuracy: 76.08%
Validation Loss: 0.6468, Validation Accuracy: 75.32%
Epoch 21/50, Train Loss: 0.6291, Train Accuracy: 76.55%
Validation Loss: 0.6487, Validation Accuracy: 75.82%
Epoch 22/50, Train Loss: 0.6295, Train Accuracy: 76.53%
Validation Loss: 0.6470, Validation Accuracy: 75.94%
Epoch 23/50, Train Loss: 0.6324, Train Accuracy: 76.60%
Validation Loss: 0.6525, Validation Accuracy: 76.08%
Epoch 24/50, Train Loss: 0.6308, Train Accuracy: 76.62%
Validation Loss: 0.6527, Validation Accuracy: 76.10%
Epoch 25/50, Train Loss: 0.6302, Train Accuracy: 76.56%
Validation Loss: 0.6507, Validation Accuracy: 76.08%
Epoch 26/50, Train Loss: 0.6285, Train Accuracy: 76.64%
Validation Loss: 0.6509, Validation Accuracy: 76.22%
Epoch 27/50, Train Loss: 0.6225, Train Accuracy: 76.81%
Validation Loss: 0.6443, Validation Accuracy: 76.24%
Epoch 28/50, Train Loss: 0.6256, Train Accuracy: 76.68%
Validation Loss: 0.6470, Validation Accuracy: 76.16%
Epoch 29/50, Train Loss: 0.6175, Train Accuracy: 77.09%
```

```
# Plot the loss curve
plt.plot(losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')  # Plot validation loss
plt.plot(accuracies, label='Training Accuracies')
plt.plot(val_accuracies, label='Validation Accuracies')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss Curve')
plt.legend()
plt.show()
```


```
from IPython.display import HTML
display(HTML('''
    <script>
        var scale = 0.5;  // 0.8 = 80% scale, adjust as needed
        document.querySelector("#notebook-container").style.transform = "scale(" + scale + ")";
        document.querySelector("#notebook-container").style.transformOrigin = "top left";
    </script>
'''))
```

Code snippets Gemini ×

"Checkin 6: Neural Networks" until only the desired sections remain visible