Forest Cover Type Prediction
CSC 461 - Final Project

Brenno Ribeiro, Emiton Alves, Matthew Silva
Professor Marco Alvarez
December 10, 2018

Introduction

This project is a study of four different machine learning algorithms on a forest cover type prediction dataset from Kaggle [1]. The four algorithms that will be used are: k-Nearest Neighbors, Logistic Regression, Support Vector Machine, and a Neural Network. Hyperparameter tuning will be performed on each algorithm along with cross-validation in order to achieve the best possible accuracy that can also generalize well out of sample. The goal of this project is to both become more familiar with these five algorithms and also to determine which one performs best on the dataset.

The dataset was taking from Kaggle.com and was used in a previously used in a competition on the website. The name of the dataset is Forest Cover Type Prediction. The objective of using this dataset is to predict forest cover types using strictly cartographic variables. The data was taken from four wilderness areas located in the Roosevelt National Forest of northern Colorado. These areas were chosen as they have experienced minimum disturbance from humans and as such the existing forest cover types are chiefly a result of ecological processes.

The first step was to analyze the dataset and become more familiar with it. This step involved looking at the various features, checking for null values, and performing some preprocessing if necessary. Next, the four algorithms were implemented starting with simple kNN up to the complex neural network. Hyperparameter selection was performed on each algorithm along with cross-validation. Finally, the algorithms were submitted on kaggle in order to measure final performance.

Data Exploration (Some Methodology, rest of methodology in each algorithms section!)

A quick study was first performed on the dataset to get a better understanding of the features present and perform any preprocessing if necessary. The training data was imported using the pandas library. The first column of the data was for the Id of the instances and as such was dropped since it would obviously have no useful effect on prediction. The dataset has 15120 instances and 55 attributes. 54 of those attributes are features of the forest areas while the last attribute is the forest cover type or label. The attributes/data fields are as follows:

Elevation - Elevation in meters
Aspect - Aspect in degrees azimuth
Slope - Slope in degrees
Horizontal_Distance_To_Hydrology - Horz Dist to nearest surface water features
Vertical_Distance_To_Hydrology - Vert Dist to nearest surface water features
Horizontal_Distance_To_Roadways - 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 solstice
Hillshade_3pm (0 to 255 index) - Hillshade index at 3pm, summer solstice
Horizontal_Distance_To_Fire_Points - 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 (7 types, integers 1 to 7) - Forest Cover Type designation

As shown above, the attributes are all categorical starting at Wilderness_Area. This means that any of kind of normalization or standardization should only be performed up to the Wilderness_Area column. The data types were then checked using the Pandas function dtypes. All attributes are of type int64. The class distribution was also checked and showed that the classes were distributed equally. Each class has exactly 2160 instances in the training data. Null values were checked using the isnull().values.any() function from Pandas and fortunately none were found. Columns with constant values were also checked for. Soil_Type7 and Soil_Type15

were shown to be constant and were therefore also dropped from the data.

Correlation between the non-categorical variables were analyzed and plotted in Fig. 1 below. As shown, there are strong negative correlations between Hillshade_9am and Hillshade_3pm, Hillshade_noon and Slope, and Aspect and Hillshade_9am. There are strong positive correlations between Horizontal_Distance_To_Hydrology and Vertical_Distance_To_Hydrology, Aspect and Hillshade_3pm, Hillshade_noon and Hillshade_3pm, and Elevation and Horizontal_Distance_To_Roadways. All of these strong correlations seem to be slightly higher than 0.6 in magnitude.

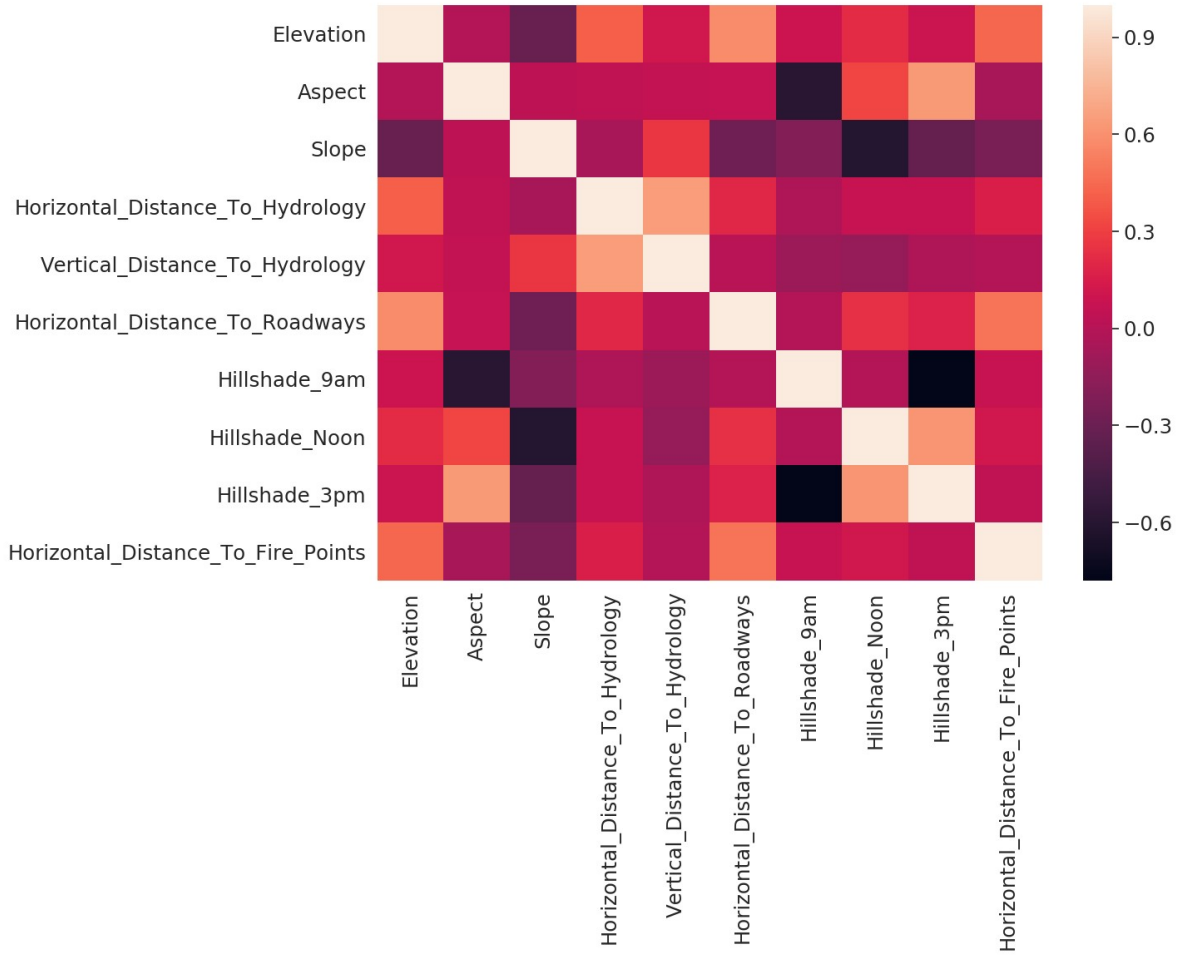

Figure 1: Correlation Heat Map

Finally the relationships between Wilderness_Area and Soil_Type to Cover_Type were studied. This was simply done by plotting Wilderness_Area vs. Cover_Type and Soil_Type vs. Cover_Type. The plots are shown in Figs. 2 and 3 respectively. As shown, wilderness area 3 has many instances but not much class distinction. Wilderness area 2 has very few instances compared to the rest. Interestingly, cover type 4 seems to only be present in wilderness area 4.

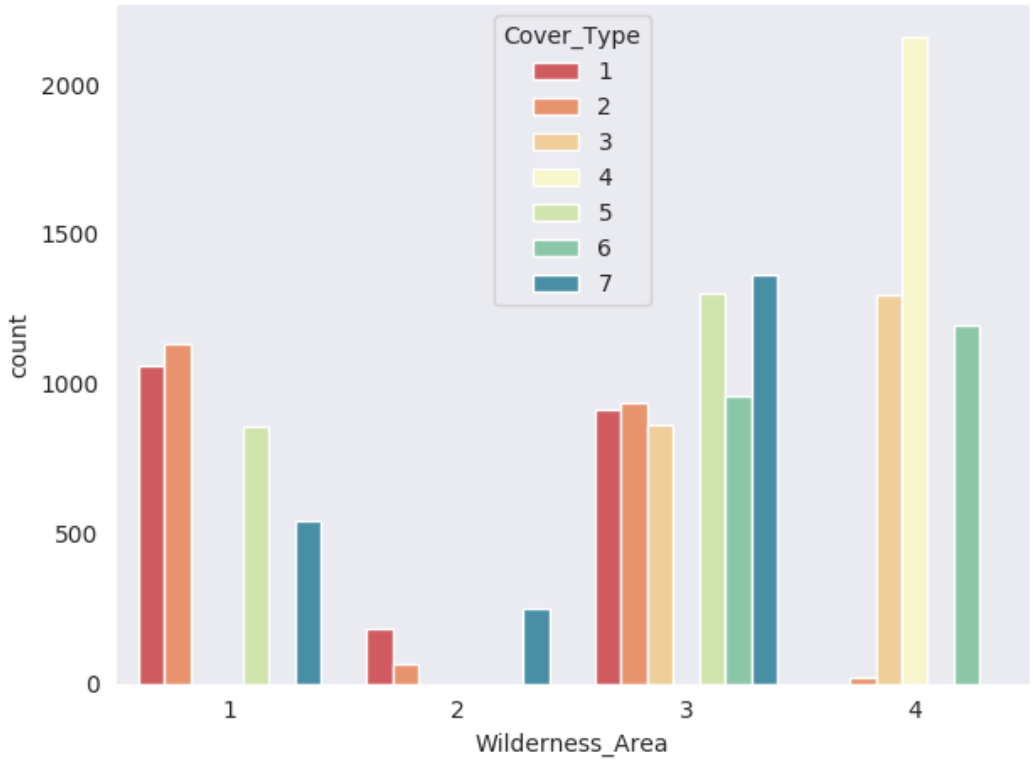

Figure 2: Wilderness Area vs. Cover Type

The plot for soil type vs. cover type is shown in Fig. 3 on the next page. It is quite easy to see that some of the soil types have great class distinction due to very high counts. Some soil types appear to be quite sparse like types 8, 9, and 26.

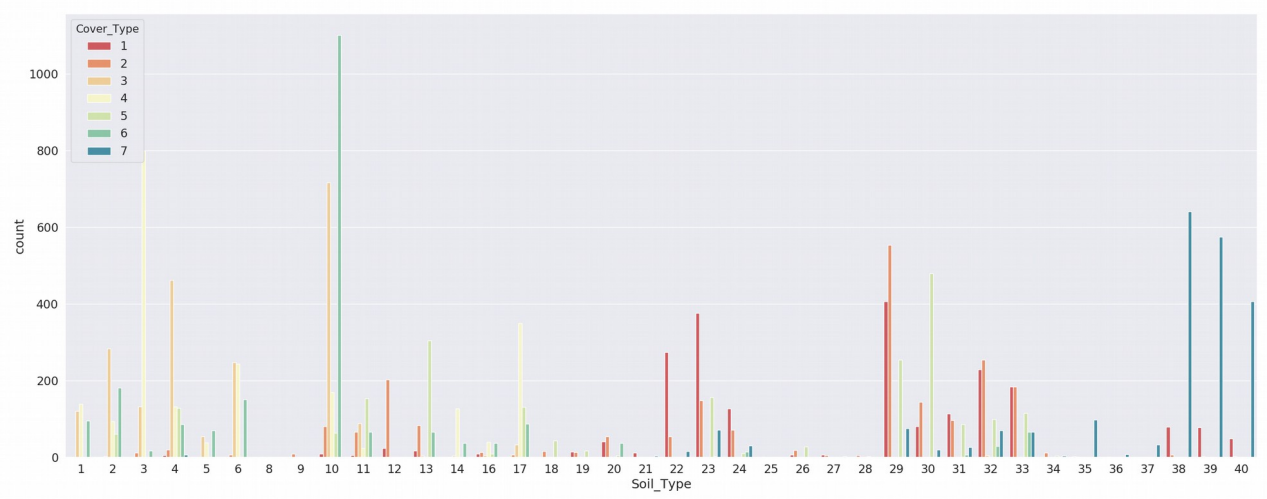

Figure 3: Soil Type vs. Cover Type

Normalization/scaling was performed on the dataset using the MinMaxScaler function from scikit-learn. This function transforms data features by scaling them to a given range. This scaling was performed on all of the non-categorical variables (up to Wilderness_Area) in the range of zero to one. Some preliminary tests showed that scaling helps tremendously come prediction time for this dataset.

k-Nearest Neighbors (Each algorithm section has Methodology and Results)

k-Nearest Neighbors (kNN) is a very simple machine learning algorithm that tends to do quite well in some cases. kNN is a form of lazy learning where all computation is performed during prediction time. The basic premise of the algorithm is that it looks at the k nearest data points to an instance during prediction time and will classify the instance based on the classes of those data points (neighbors). This algorithm was implemented using the kNeighborsClassifier class from scikit-learn. Three hyperparameters were taken into consideration: number of

neighbors, weight metric, and p. The number of neighbors is the number of nearest data points that are used during computation. The weight metric is important for classification as it decides how each neighbor should be treated in terms of influence on prediction. The weight metric is 'uniform' by default which means that the class of an instance to predicted is determined by a majority vote of the classes of the neighbors. Another form of weighting was taken a look at, 'distance', which gives more influence to the classes of closer data points. The last hyperparameter is p, which dictates what form of distance metric is used during computation. Values of 1 and 2 were considered for p which correspond to manhattan and euclidean distances respectively.

The first two tests involved varying the number of neighbors between 3 and 99 for p values of 1 and 2 while fixing the weight metric at 'uniform'. The results are shown below in Figs. 4 and 5. The accuracies were found by taking the average accuracy from 5 fold cross validation tests.

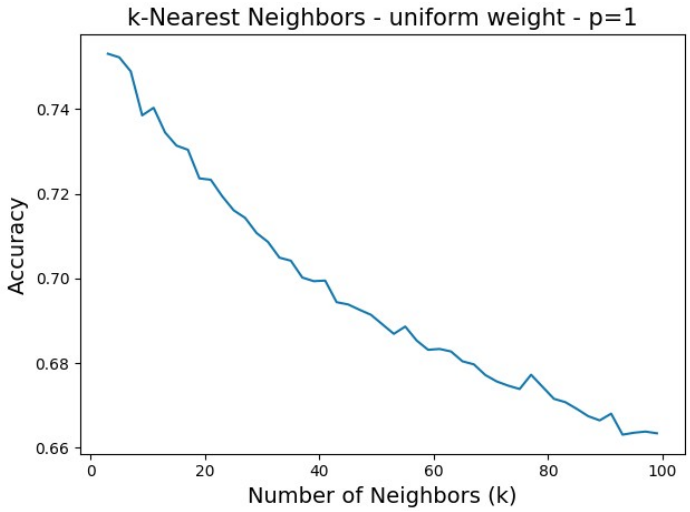

Figure 4: Number of Neighbors vs. Accuracy, uniform weight, p = 1

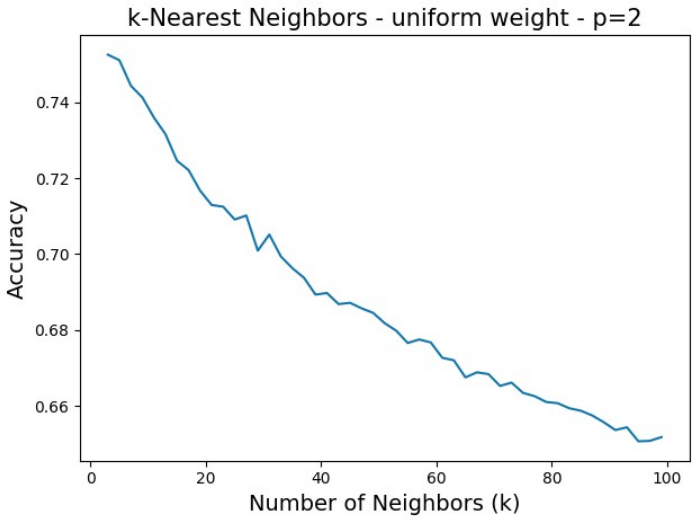

Figure 5: Number of Neighbors vs. Accuracy, uniform weight, p = 2

As shown above, lower number of neighbors performed significantly better than higher numbers for both cases. The best accuracy was 75.30 % and was achieved with 3 neighbors and a p value of 1. Two additional tests were performed with the exact same conditions but the weight metric fixed at ‘distance’. The results are shown below in Figs. 6 and 7. The accuracies were once again found by taking the average accuracy from 5 fold cross validation tests.

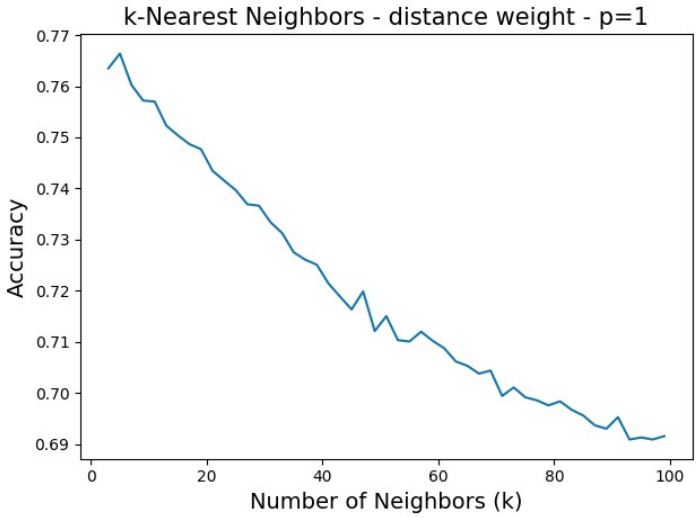

Figure 6: Number of Neighbors vs. Accuracy, distance weight, p = 1

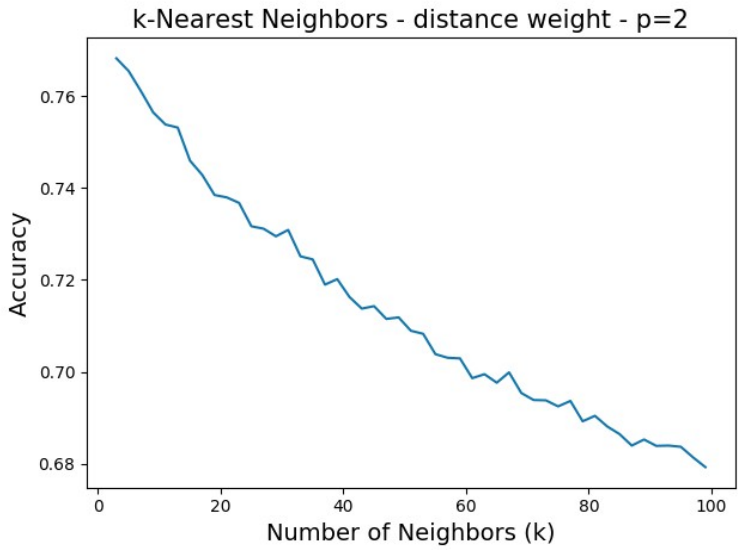

Figure 7: Number of Neighbors vs. Accuracy, distance weight, p = 2

Once again, a lower number of neighbors performed significantly better than higher numbers. The best accuracy was 76.83 % and was achieved with 3 neighbors and a p value of 2. This configuration of distance weight metric, 3 neighbors, and p value of 2 was the best out of all the tests. This an extremely satisfying result for such a simple algorithm that was so quick and easy to implement. The confusion matrix for the best hyperparameters is shown below in Fig. 8.

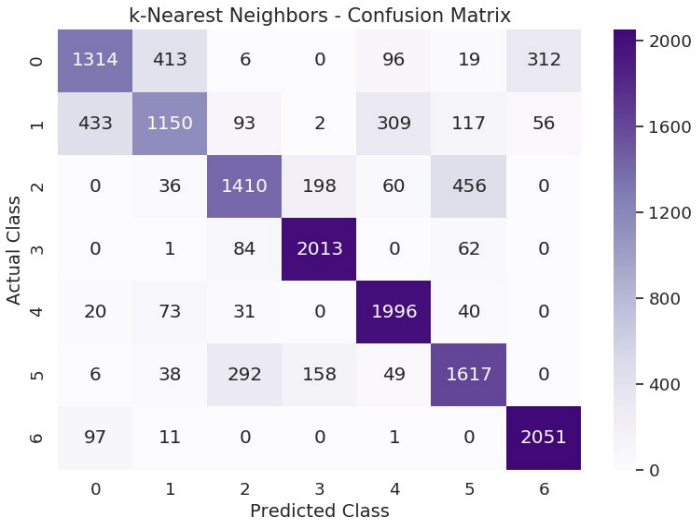

Figure 8: k-Nearest Neighbors - Confusion Matrix

Logistic Regression

Logistic Regression is a popular algorithm used in many machine learning applications. For this particular dataset, it was believed that logistic regression would serve well for multiclass classification. One of the many benefits of logistic regression is that the probability of the instance belonging to a class is determined. This is useful for critical applications where it is important to understand how likely it is that a data instance belongs to a class. There were two main hyperparameters that chosen for tuning. The first parameter was the 'C' value. C is actually the inverse of lamda, a regularization parameter. The C parameter affects the strength of regularization. Higher values of C will correlate to less regularization, while lower values of C will correlate to more regularization. Lower regularization strength will create more complex models which can overfit the data, whereas higher regularization strength will create simpler models which can underfit the data. The second hyperparameter was the penalty. The two types of penalty chosen for tuning was the L1 and L2 penalty. L1 penalty will add the absolute value of magnitude coefficient to the loss function, where L2 will add the squared magnitude of the same value to the loss function.

In order to tune the hyper parameters, multiple tests were run using different values of C and either L1 or L2. The initial values of C used for testing started with .001 and increased logarithmically. Regardless of the values of C, it was seen that a L1 penalty scheme produced better results than using L2 penalties. Additionally, the accuracy of the model increased as the value of C increased. This suggests that less regularization can lead to better prediction for this given model. When regularization was high, given a small C value, the model performed poorly at 57 % accuracy. Increasing C led to a more than 10 % increase. Though there seems to be a theoretic bound to the influence of C. Increasing C by factors of 10 do very little to help the

model once the accuracy is in the range of 68%. When C is within the range of 10 to 1,000,000 the largest change in accuracy is roughly 0.11%. In this case C = 10 produces the best accuracy. Additionally, using high values of C can lead to overfitting. The best hyperparameter pair was a L1 penalty scheme and C = 10. The results from the tests are shown below in Table 1.

C	0.001	0.01	0.1	1	10	100	1000	10000	100000
Accuracy	56.59 %	61.32 %	67.14 %	67.99 %	68.44 %	68.37%	68.33 %	68.36 %	68.41 %

Table 1: Logistic Regression - C vs Accuracy

The confusion matrix is shown below in Fig. 9.

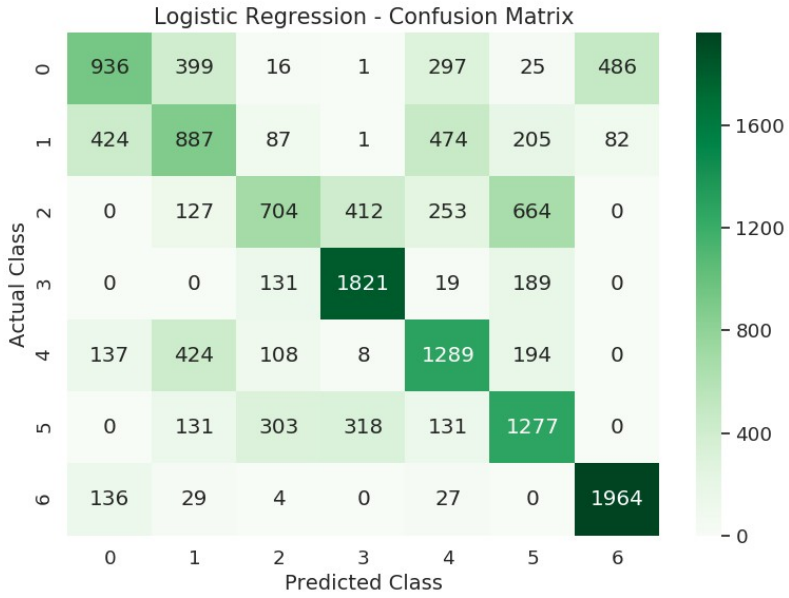

Figure 9: Logistic Regression - Confusion Matrix

Support Vector Machine

Support vector machine is a very popular machine learning algorithm that has been used extensively in research. This algorithm also tends to perform extremely well in practice and as such was chosen for testing with this dataset. The algorithm was implemented using the SVC class from scikit-learn. Three hyperparameters were considered for tuning. Those parameters were C (inverse of regularization parameter lambda), the kernel, and the degree for the polynomial kernel. C is a very important parameter used for regularization. Low values of C allow more points to be misclassified while higher values have a tighter margin and as such the algorithm attempts to fit the data quite heavily. Two kernels were considered: radial basis function/gaussian and polynomial. The polynomial kernel had an additional hyperparameter to define the number of degrees in the polynomial.

The first test used the gaussian kernel and varied the value of C. The results are shown below in Table 2.

C	0.001	0.01	0.1	1	10	100	1000	10000	100000
Accuracy	42.22 %	42.20 %	53.06 %	64.60 %	68.09 %	70.37 %	72.80 %	74.44 %	76.11 %

Table 2: Gaussian Kernel - C vs. Accuracy

As shown above, higher values of C performed significantly better than lower values. This indicates that the model did not have any problems with overfitting. Tests were run using a polynomial kernel with degrees ranging from 2 to 5. C was only varied between 1 to 100000 since the gaussian kernel results clearly showed that higher values performed much better. The results for the polynomial kernel tests are shown below in Table 3.

C		1	10	100	1000	10000	100000
Accuracy	**Degree 2**	61.14 %	66.39 %	69.72 %	71.34 %	73.10 %	73.51 %
	Degree 3	51.52 %	64.91 %	69.22 %	71.28 %	73.85 %	75.95 %
	Degree 4	35.98 %	61.29 %	67.84 %	71.92 %	73.35 %	75.60 %
	Degree 5	29.07 %	47.61 %	64.84 %	69.82 %	72.24 %	75.13 %

Table 3: Polynomial Kernel - C vs. Degree vs. Accuracy

As shown above, a polynomial kernel with degree 3 and a C value of 100000 performed the best with an accuracy of 75.95 %. Similar to the gaussian kernel, higher values of C performed the best once again. This confirms once again that the algorithm is simply not having any difficulties with overfitting and that regularization is not needed on this dataset when using support vector machines. The best hyperparameter combination was the gaussian kernel with a C value of 100000. This combination achieved an accuracy of 76.11 %. The confusion matrix for this configuration on 5-fold cross validation is shown below in Fig. 10.

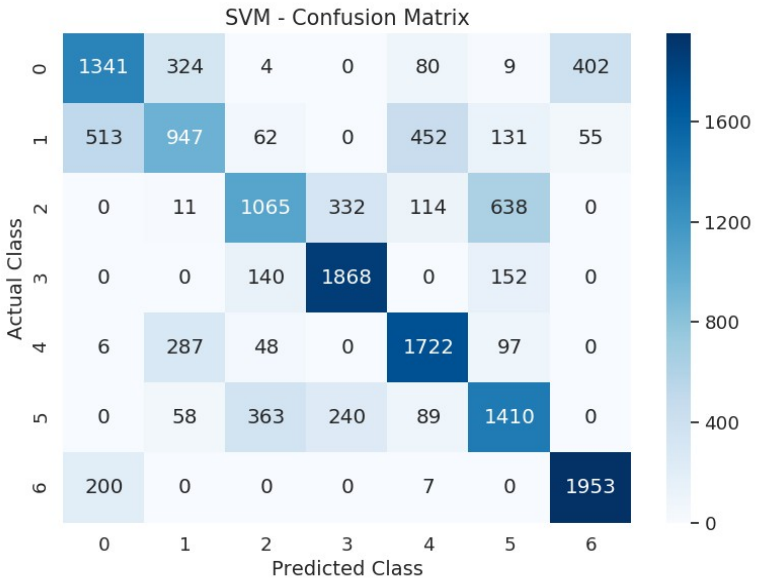

Fig 10: SVM - Confusion Matrix

Neural Network

Neural networks are a biologically-inspired programming paradigm that has significantly increased in popularity over the last few years [2]. Neural networks are at the center of an ongoing deep learning revolution. A simple feed-forward neural network with one hidden layer was considered for study on this dataset. The neural network was implemented using Google's open source machine learning framework, Tensorflow [3]. Neural networks can be quite computationally intensive to implement so a GPU (Nvidia Geforce GTX 1080) was used for computations. This was done using Nvidia's CUDA platform [4], a programming model used for computing on GPUs.

The neural network has two layers, one hidden and one output. The activation functions used in the hidden layer were rectified linear units. This function outputs zero if its input is negative and is linear if the input is zero or greater. The output layer utilized a softmax to output probabilities for classification. Cross entropy error was used as a cost function and the AdamOptimizer was used to minimize the error. Adam is an extension to stochastic gradient that takes advantage of momentum terms to minimize the cost function. It is a very popular optimizer used in research. The labels of the training data was one hot encoded prior to use of the neural network in order to use a softmax for prediction. A diagram of the network is given on the next page in Fig. 11.

The only hyperparameters taken into account for the neural network were the number of nodes in the hidden layer and the batch size of the adam optimizer. The number of nodes was varied between 10 and 50. There is evidence to suggest that small mini batch sizes are better for training [5], as such the batch size was varied between 2 and 32 for testing. Tests on the batch size were done first with a fixed hidden layer size of 50 nodes and a fixed set of 1000 epochs.

The results are shown below in Table 4.

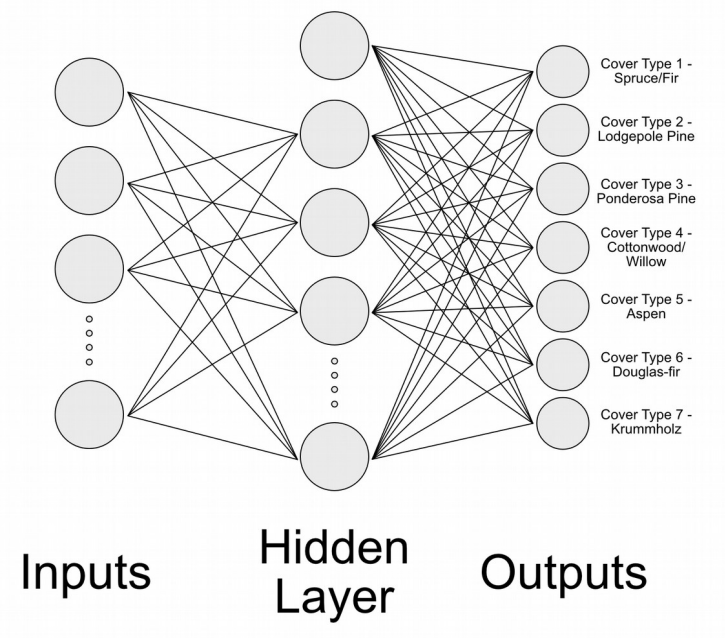

Figure 11: Neural Network Diagram

Batch Size	2	4	8	16	32
Accuracy	76.79 %	80.02 %	79.78 %	79.56 %	81.35 %

Table 4: Batch Size vs. Accuracy

As shown above, a batch size of 32 was the best with a test accuracy of 81.35 %. The batch size was then set to 32 and the only hyperparameter left to tune was the number of nodes in the first layer. The number of nodes was varied between 10 to 50 with the previously found batch size. Epochs were fixed at 1000 like the last test. The results are shown in Table 5 on the next page.

Nodes	10	15	20	25	30	35	40	45	50
Accuracy	78.88 %	78.59 %	78.48 %	79.66 %	80.44 %	79.00 %	80.85 %	80.69 %	81.35 %

Table 5: Number of Hidden Layer Nodes vs. Accuracy

As shown above 50 nodes performed the best with an accuracy of 81.35 %. The neural network model chosen was then the two layer neural network with 50 nodes in the hidden layer and a batch size of 32. This configuration performed extremely well and was the best out of all the algorithms used. The confusion matrix for this configuration is shown below in Fig. 12.

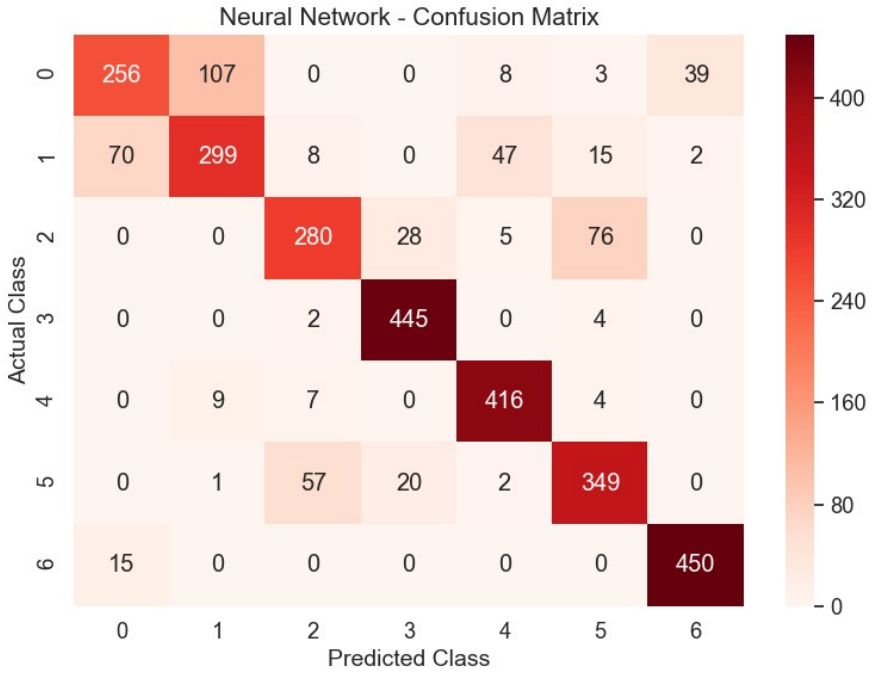

Figure 12: Neural Network – Confusion Matrix

Conclusions and Test Submission

The neural network performed the best out of the four algorithms with an accuracy of 81.35 %. k-Nearest Neighbors performed surprisingly well achieving an accuracy of 76.83 % with three neighbors, distance weighting, and p = 2. Logistic regression was quite disappointing, only achieving an accuracy of 68.44 % with a L1 penalty and C value of 10. Support vector machines performed fairly well with a final accuracy of 76.11 % when using the gaussian kernel and a large C value of 100000.

Since the neural network performed the best in training, it was chosen for the first test submission to Kaggle. The final configuration of the two layer neural network was a hidden layer with 50 nodes and a batch size of 32 with the adam optimizer. A csv file with instance Id and cover type prediction was written using the provided test data from Kaggle. This file was submitted to Kaggle to test the final performance. The submission achieved an accuracy of 65.57 %! This was a very dissapointing result and was simply not expected. It is possible that the neural network overfitted the data and that's why it performed so badly.

The other three algorithms were submitted to Kaggle in the same manner. Each algorithm used their respective best hyperparameter configuration. k-Nearest Neighbors achieved an accuracy of 65.58 %. Logistic Regression achieved an accuracy of 52.70 %. The support vector machine achieved an accuracy of 67.67%. Logistic regression was dissapoint once again as expected from the cross-validation results. k-Nearest Neighbors performed extremely well once again for such a simple algorithm. kNN matched the neural network performance which took significantly longer to train and tune. The support vector machine performed the best out of all the algorithms.

The results were very unexpected but extremely interesting. It is possible that overfitting

occured even though cross-validation was used. The results obtained were not good enough to place well on the leaderboard. While the results were not as satisfactory as expected, this project was a great introduction to using different types of machine learning algorithms on a real world dataset. All group members significantly improved their understanding of the algorithms used and their compentence of implementing these models in Python. Further work could be done to better fit the data and possibly achieve better performance results. It is possible that other algorithms and methods would be better suited to this dataset than the ones used in this study. Ensemble methods could be examined for example. In conclusion, much was learned from performing this study but further work would have to be done in order to improve the results obtained on Kaggle.

Source Code

k-Nearest Neighbors, Logistic Regression, and Support Vector Machines were all implemented using scikit-learn. As such, a script was written to facilitate cross-validation for the use of those algorithms. This script contained functions for initializing data and testing the models using cross validation. Loops were created with these functions in order to tune hyperparameters. The script is shown below:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
```

```
def init_data(normalize=True):
    """
    Initializes the data.
    """
    data = pd.read_csv('train.csv')
    data = data.drop(['Id', 'Soil_Type7', 'Soil_Type15'],
axis=1)

    X = data.iloc[:, :-1].values
    y = data.iloc[:, -1].values

    if normalize:
        min_max_scaler = MinMaxScaler()
        x2 = X[:, 10:]
        x1 = min_max_scaler.fit_transform(X[:, :10])
        X = np.concatenate((x1, x2), axis=1)

    return X, y


actual_list = []
predict_list = []


def model_analysis(model, X, y):
    """
    Perform cross validation and score analysis for different
algorithms.
    """
    model = model

    print("=" * 100, "\n", str(model), "\n", "=" * 100)

    skf = StratifiedKFold(n_splits=5, shuffle=True)

    accuracies = []

    # Cross-Validation
```

```
    for train_index, test_index in skf.split(X, y):
        X_train = X[train_index]
        X_test = X[test_index]
        y_train = y[train_index]
        y_test = y[test_index]

        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

        score = accuracy_score(y_test, y_pred)
        accuracies.append(score)
        actual_list.append(list(y_test))
        predict_list.append(list(y_pred))

    accuracy = np.average(accuracies)
    print("Accuracy Score:", accuracy)


X, y = init_data()

# Hyperparameter Testing
for C in [0.01, 0.1, 1, 10, 100]:
    svm = SVC(C=C, kernel='rbf')
    model_analysis(svm, X, y)
```

The confusion matrices for the first three algorithms were plotted using seaborn. The code is shown below:

```
actual_list = sum(actual_list, [])
predict_list = sum(predict_list, [])
cm = confusion_matrix(actual_list, predict_list)

# Confusion Matrix Heatmap

plt.figure(figsize=(8,6))
sns.set(font_scale=1.2)
sns.heatmap(cm, annot=True, cmap='Greens', fmt='g')
plt.xlabel('Predicted Class', fontsize=14)
plt.ylabel('Actual Class', fontsize=14)
```

```
plt.title('Logistic Regression - Confusion Matrix', fontsize=15)
plt.show()
```

The neural network was done separately from the other algorithms as it used tensorflow instead of scikit-learn. Hyperparameter tuning was done by simply changing parameter values and measuring resulting accuracies. The script is shown on the next page:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import tensorflow as tf

data = pd.read_csv('train.csv')
data = data.drop(['Id', 'Soil_Type7', 'Soil_Type15'], axis=1)

X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values
y = pd.get_dummies(y).values

min_max_scaler = MinMaxScaler()
x2 = X[:, 10:]
x1 = min_max_scaler.fit_transform(X[:, :10])
X = np.concatenate((x1, x2), axis=1)

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

model = tf.keras.models.Sequential()

# Input Layer
model.add(tf.keras.layers.Dense(52, activation='relu',
input_shape=(X.shape[1],)))

# Hidden Layer
model.add(tf.keras.layers.Dropout(0.3, noise_shape=None,
seed=None))
```

```
model.add(tf.keras.layers.Dense(50, activation='relu'))
model.add(tf.keras.layers.Dropout(0.2, noise_shape=None,
seed=None))

# Output Layer
model.add(tf.keras.layers.Dense(7, activation='softmax'))

model.compile(optimizer="adam", loss='categorical_crossentropy',
metrics=["accuracy"])
results = model.fit(X_train, y_train, epochs=1000,
batch_size=32, validation_data=(X_test, y_test))
print("Accuracy:", np.mean(results.history["val_acc"]))
```

Kaggle submissions had to be formatted as a csv file before being submitted. The script used for setting up the data for this file is shown on the next page:

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
import csv
import tensorflow as tf

train = pd.read_csv('train.csv')
train = train.drop(['Id', 'Soil_Type7', 'Soil_Type15'], axis=1)

test = pd.read_csv('test.csv')
Id = test['Id']
test = test.drop(['Id', 'Soil_Type7', 'Soil_Type15'], axis=1)

X_train = train.iloc[:, :-1].values
y_train = train.iloc[:, -1].values
# y_train = pd.get_dummies(y_train).values      # One-Hot
```

```
Encoding for Neural Network

X_test = test.iloc[:, :].values

min_max_scaler = MinMaxScaler()
x2_train = X_train[:, 10:]
x1_train = min_max_scaler.fit_transform(X_train[:, :10])
X_train = np.concatenate((x1_train, x2_train), axis=1)

x2_test = X_test[:, 10:]
x1_test = min_max_scaler.fit_transform(X_test[:, :10])
X_test = np.concatenate((x1_test, x2_test), axis=1)
```

The part of the code to write to the csv is shown below:

```
lr = LogisticRegression(C=10, penalty='l1')
lr.fit(X_train, y_train)
predict = lr.predict(X_test)

predictions = []
for i in range(len(predict)):
    predictions.append([Id[i], predict[i]])

with open('submission_lr_new.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerows(predictions)
```

The neural network required a different approach due to the one hot encoding. The code is shown below:

```
model = tf.keras.models.Sequential()

# Input Layer
model.add(tf.keras.layers.Dense(52, activation='relu',
input_shape=(X_train.shape[1],)))

# Hidden Layer
model.add(tf.keras.layers.Dropout(0.3, noise_shape=None,
seed=None))
```

```
model.add(tf.keras.layers.Dense(50, activation='relu'))
model.add(tf.keras.layers.Dropout(0.2, noise_shape=None,
seed=None))

# Output Layer
model.add(tf.keras.layers.Dense(7, activation='softmax'))

model.compile(optimizer="adam", loss='categorical_crossentropy',
metrics=["accuracy"])
results = model.fit(X_train, y_train, epochs=50, batch_size=32,
verbose=False)
predict = model.predict(X_test)

predictions = []
for i in range(len(predict)):
    predictions.append([Id[i], np.argmax(predict[i]) + 1])

with open('submission_lr_new.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerows(predictions)
```

References

[1] Kaggle. (2014). *Forest Cover Type Prediction*. Available at: https://www.kaggle.com/c/forest-cover-type-prediction.

[2] Hardesty, Larry. “Explained: Neural Networks.” MIT News, 14 Apr. 2017, news.mit.edu/2017/explained-neural-networks-deep-learning-0414.

[3] TensorFlow, www.tensorflow.org/.

[4] “About CUDA.” NVIDIA Developer, 11 Oct. 2018, developer.nvidia.com/about-cuda.

[5] Masters, Dominic, and Carlo Luschi. “Revisiting Small Batch Training for Deep Neural Networks.” Graphcore, Apr. 2018, www.graphcore.ai/posts/revisiting-small-batch-training-for-deep-neural-networks.