Nonparametric ML Models - Cumulative Lab

Introduction

In this cumulative lab, you will apply two nonparametric models you have just learned — k-nearest neighbors and decision trees — to the forest cover dataset.

Objectives

- Practice identifying and applying appropriate preprocessing steps
- Perform an iterative modeling process, starting from a baseline model
- Explore multiple model algorithms, and tune their hyperparameters
- Practice choosing a final model across multiple model algorithms and evaluating its performance

Your Task: Complete an End-to-End ML Process with Nonparametric Models on the Forest Cover Dataset

Photo by Michael Benz (https://unsplash.com/@michaelbenz?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on Unsplash (/s/photos/forest?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText)

Business and Data Understanding

To repeat the previous description:

Here we will be using an adapted version of the forest cover dataset from the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/covertype). Each record represents a 30 x 30 meter cell of land within Roosevelt National Forest in northern Colorado, which has been labeled as `Cover_Type` 1 for "Cottonwood/Willow" and `Cover_Type` 0 for "Ponderosa Pine". (The original dataset contained 7 cover types but we have simplified it.)

In [1]:
```
# Run this cell without changes
import pandas as pd

df = pd.read_csv('data/forest_cover.csv')
df
```

Out[1]:

	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horiz
0	2553	235	17	351	95	
1	2011	344	17	313	29	
2	2022	24	13	391	42	
3	2038	50	17	408	71	
4	2018	341	27	351	34	
...	...	...	...	...	...	
38496	2396	153	20	85	17	
38497	2391	152	19	67	12	
38498	2386	159	17	60	7	
38499	2384	170	15	60	5	
38500	2383	165	13	60	4	

38501 rows × 53 columns

As you can see, we have over 38,000 rows, each with 52 feature columns and 1 target column:

- `Elevation` : Elevation in meters
- `Aspect` : Aspect in degrees azimuth
- `Slope` : Slope in degrees

In [2]:
```
# Run this cell without changes
print("Raw Counts")
print(df["Cover_Type"].value_counts())
print()
print("Percentages")
print(df["Cover_Type"].value_counts(normalize=True))
```

```
Raw Counts
0    35754
1     2747
Name: Cover_Type, dtype: int64

Percentages
0    0.928651
1    0.071349
Name: Cover_Type, dtype: float64
```

Thus, a baseline model that always chose the majority class would have an accuracy of over 92%. Therefore we will want to report additional metrics at the end.

Previous Best Model

In a previous lab, we used SMOTE to create additional synthetic data, then tuned the hyperparameters of a logistic regression model to get the following final model metrics:

- **Log loss:** 0.13031294393913376
- **Accuracy:** 0.9456679825472678
- **Precision:** 0.6659919028340081
- **Recall:** 0.47889374090247455

In this lab, you will try to beat those scores using more-complex, nonparametric models.

Modeling

Although you may be aware of some additional model algorithms available from scikit-learn, for this lab you will be focusing on two of them: k-nearest neighbors and decision trees. Here are some reminders about these models:

kNN - documentation here (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html)

This algorithm — unlike linear models or tree-based models — does not emphasize learning the relationship between the features and the target. Instead, for a given test record, it finds the most similar records in the training set and returns an average of their target values.

- **Training speed:** Fast. In theory it's just saving the training data for later, although the scikit-learn implementation has some additional logic "under the hood" to make prediction faster.
- **Prediction speed:** Very slow. The model has to look at every record in the training set to find the k closest to the new record.
- **Requires scaling:** Yes. The algorithm to find the nearest records is distance-based, so it matters that distances are all on the same scale.

- **Key hyperparameters:** `n_neighbors` (how many nearest neighbors to find; too few neighbors leads to overfitting, too many leads to underfitting), `p` and `metric` (what kind of distance to use in defining "nearest" neighbors)

Decision Trees - documentation here (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html)

Similar to linear models (and unlike kNN), this algorithm emphasizes learning the relationship between the features and the target. However, unlike a linear model that tries to find linear relationships between each of the features and the target, decision trees look for ways to split the data based on features to decrease the entropy of the target in each split.

- **Training speed:** Slow. The model is considering splits based on as many as all of the available features, and it can split on the same feature multiple times. This requires exponential computational time that increases based on the number of columns as well as the number of rows.
- **Prediction speed:** Medium fast. Producing a prediction with a decision tree means applying several conditional statements, which is slower than something like logistic regression but faster than kNN.
- **Requires scaling:** No. This model is not distance-based. You also can use a `LabelEncoder` rather than `OneHotEncoder` for categorical data, since this algorithm doesn't necessarily assume that the distance between `1` and `2` is the same as the distance between `2` and `3`.
- **Key hyperparameters:** Many features relating to "pruning" the tree. By default they are set so the tree can overfit, and by setting them higher or lower (depending on the hyperparameter) you can reduce overfitting, but too much will lead to underfitting. These are: `max_depth`, `min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf`, `max_features`, `max_leaf_nodes`, and `min_impurity_decrease`. You can also try changing the `criterion` to "entropy" or the `splitter` to "random" if you want to change the splitting logic.

Requirements

1. Prepare the Data for Modeling

2. Build a Baseline kNN Model

3. Build Iterative Models to Find the Best kNN Model

4. Build a Baseline Decision Tree Model

5. Build Iterative Models to Find the Best Decision Tree Model

6. Choose and Evaluate an Overall Best Model

1. Prepare the Data for Modeling

The target is `Cover_Type`. In the cell below, split `df` into `X` and `y`, then perform a train-test split with `random_state=42` and `stratify=y` to create variables with the standard `X_train`, `X_test`, `y_train`, `y_test` names.

Include the relevant imports as you go.

In [3]:
```
# Your code here

# Import the relevant function
from sklearn.model_selection import train_test_split

# Split df into X and y
X = df.drop("Cover_Type", axis=1)
y = df["Cover_Type"]

# Perform train-test split with random_state=42 and stratify=y
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, stratify=y
```

Now, instantiate a `StandardScaler`, fit it on `X_train`, and create new variables `X_train_scaled` and `X_test_scaled` containing values transformed with the scaler.

In [4]:
```
# Your code here

# Import the relevant class
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(X_train)

X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

The following code checks that everything is set up correctly:

In [5]:
```
# Run this cell without changes

# Checking that df was separated into correct X and y
assert type(X) == pd.DataFrame and X.shape == (38501, 52)
assert type(y) == pd.Series and y.shape == (38501,)

# Checking the train-test split
assert type(X_train) == pd.DataFrame and X_train.shape == (28875, 52)
assert type(X_test) == pd.DataFrame and X_test.shape == (9626, 52)
assert type(y_train) == pd.Series and y_train.shape == (28875,)
assert type(y_test) == pd.Series and y_test.shape == (9626,)

# Checking the scaling
assert X_train_scaled.shape == X_train.shape
assert round(X_train_scaled[0][0], 3) == -0.636
assert X_test_scaled.shape == X_test.shape
assert round(X_test_scaled[0][0], 3) == -1.370
```

2. Build a Baseline kNN Model

Build a scikit-learn kNN model with default hyperparameters. Then use `cross_val_score` with `scoring="neg_log_loss"` to find the mean log loss for this model (passing in `X_train_scaled` and `y_train` to `cross_val_score`). You'll need to find the mean of the cross-validated scores, and negate the value (either put a `-` at the beginning or multiply by `-1`) so that your answer is a log loss rather than a negative log loss.

Call the resulting score `knn_baseline_log_loss`.

Your code might take a minute or more to run.

In [6]:
```
# Replace None with appropriate code

# Relevant imports
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score

# Creating the model
knn_baseline_model = KNeighborsClassifier()

# Perform cross-validation
knn_baseline_log_loss = -cross_val_score(knn_baseline_model, X_train_scaled, y_train,
knn_baseline_log_loss
```

Out[6]:
```
0.1255288892455634
```

Our best logistic regression model had a log loss of 0.13031294393913376

Is this model better? Compare it in terms of metrics and speed.

In [ ]:
```
# Replace None with appropriate text
"""
Our log loss is better with the vanilla (un-tuned) kNN model
than it was with the tuned logistic regression model

It was also much slower, taking around a minute to complete
the cross-validation on this machine

It depends on the business case whether this is really a better
model
"""
```

3. Build Iterative Models to Find the Best kNN Model

Build and evaluate at least two more kNN models to find the best one. Explain why you are changing the hyperparameters you are changing as you go. These models will be *slow* to run, so be thinking about what you might try next as you run them.

In [7]:
```
# Your code here (add more cells as needed)
knn_third_model = KNeighborsClassifier(n_neighbors=50, metric="manhattan")

knn_third_log_loss = -cross_val_score(knn_third_model, X_train_scaled, y_train, scori
knn_third_log_loss
```

Out[7]:
```
0.07621145166565102
```

In [ ]:
```
# Your code here (add more cells as needed)
knn_fourth_model = KNeighborsClassifier(n_neighbors=75, metric="manhattan")

knn_fourth_log_loss = -cross_val_score(knn_fourth_model, X_train_scaled, y_train, sco
knn_fourth_log_loss
```

```
the third model is the most optimal.
```

4. Build a Baseline Decision Tree Model

Now that you have chosen your best kNN model, start investigating decision tree models. First, build and evaluate a baseline decision tree model, using default hyperparameters (with the exception of `random_state=42` for reproducibility).

(Use cross-validated log loss, just like with the previous models.)

In [8]:
```
# Your code here
from sklearn.tree import DecisionTreeClassifier

dtree_baseline_model = DecisionTreeClassifier(random_state=42)

dtree_baseline_log_loss = -cross_val_score(dtree_baseline_model, X_train, y_train, sc
dtree_baseline_log_loss
```

Out[8]:
```
0.7045390124149022
```

Interpret this score. How does this compare to the log loss from our best logistic regression and best kNN models? Any guesses about why?

In [ ]:
```
# Replace None with appropriate text
"""
This is much worse than either the logistic regression or the
kNN models. We can probably assume that the model is badly
overfitting, since we have not "pruned" it at all.
"""
```

5. Build Iterative Models to Find the Best Decision Tree Model

Build and evaluate at least two more decision tree models to find the best one. Explain why you are changing the hyperparameters you are changing as you go.

In [9]:
```
# Your code here (add more cells as needed)
dtree_second_model = DecisionTreeClassifier(random_state=42, min_samples_leaf=10)

dtree_second_log_loss = -cross_val_score(dtree_second_model, X_train, y_train, scorin
dtree_second_log_loss
```

Out[9]:
```
0.28719567271672036
```

In [10]:
```
# Your code here (add more cells as needed)
dtree_third_model = DecisionTreeClassifier(random_state=42, min_samples_leaf=100)

dtree_third_log_loss = -cross_val_score(dtree_third_model, X_train, y_train, scoring=
dtree_third_log_loss
```

Out[10]:
```
0.11894015917756935
```

In [11]:
```
# Your code here (add more cells as needed)
dtree_fourth_model = DecisionTreeClassifier(random_state=42, min_samples_leaf=100, cl

dtree_fourth_log_loss = -cross_val_score(dtree_fourth_model, X_train, y_train, scorin
dtree_fourth_log_loss
```

Out[11]:
```
0.20808868577091694
```

6. Choose and Evaluate an Overall Best Model

Which model had the best performance? What type of model was it?

Instantiate a variable `final_model` using your best model with the best hyperparameters.

In [12]:
```
# Replace None with appropriate code
final_model =  KNeighborsClassifier(n_neighbors=50, metric="manhattan")

# Fit the model on the full training data
# (scaled or unscaled depending on the model)
final_model.fit(X_train_scaled, y_train)
```

Out[12]:
```
KNeighborsClassifier(metric='manhattan', n_neighbors=50)
```

Now, evaluate the log loss, accuracy, precision, and recall. This code is mostly filled in for you, but you need to replace `None` with either `X_test` or `X_test_scaled` depending on the model you chose.

In [14]:
```
# Replace None with appropriate code
from sklearn.metrics import accuracy_score, precision_score, recall_score,log_loss

preds = final_model.predict(X_test_scaled)
probs = final_model.predict_proba(X_test_scaled)

print("log loss: ", log_loss(y_test, probs))
print("accuracy: ", accuracy_score(y_test, preds))
print("precision:", precision_score(y_test, preds))
print("recall:   ", recall_score(y_test, preds))
```

```
log loss:  0.07491819679665564
accuracy:  0.9716393102015375
precision: 0.8876404494382022
recall:    0.6899563318777293
```

Interpret your model performance. How would it perform on different kinds of tasks? How much better is it than a "dummy" model that always chooses the majority class, or the logistic regression described at the start of the lab?

In [ ]:

```
# Replace None with appropriate text
"""
This model has 97% accuracy, meaning that it assigns the
correct label 97% of the time. This is definitely an
improvement over a "dummy" model, which would have about
92% accuracy.

If our model labels a given forest area a 1, there is
about an 89% chance that it really is class 1, compared
to about a 67% chance with the logistic regression

The recall score is also improved from the logistic
regression model. If a given cell of forest really is
class 1, there is about a 69% chance that our model
will label it correctly. This is better than the 48%
of the logistic regression model, but still doesn't
instill a lot of confidence. If the business really
cared about avoiding "false negatives" (labeling
cottonwood/willow as ponderosa pine) more so than
avoiding "false positives" (labeling ponderosa pine
as cottonwood/willow), then we might want to adjust
the decision threshold on this
"""
```

Conclusion

In this lab, you practiced the end-to-end machine learning process with multiple model algorithms, including tuning the hyperparameters for those different algorithms. You saw how nonparametric models can be more flexible than linear models, potentially leading to overfitting but also potentially reducing underfitting by being able to learn non-linear relationships between variables. You also likely saw how there can be a tradeoff between speed and performance, with good metrics correlating with slow speeds.