index

March 27, 2022

Link is here or:

`https://github.com/miladshiraniUCB/dsc-nonparametric-models-lab.git`

1 Nonparametric ML Models - Cumulative Lab

1.1 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.

1.2 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

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

Photo by Michael Benz on Unsplash

1.3.1 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. 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.)

The task is to predict the `Cover_Type` based on the available cartographic variables:

[3]:
```
# Run this cell without changes
import pandas as pd

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

[3]:
```
      Elevation  Aspect  Slope  Horizontal_Distance_To_Hydrology  \
0          2553     235     17                               351
1          2011     344     17                               313
2          2022      24     13                               391
```

```
3           2038      50     17                                408
4           2018     341     27                                351
…           …     …    …                                    …
38496       2396     153     20                                 85
38497       2391     152     19                                 67
38498       2386     159     17                                 60
38499       2384     170     15                                 60
38500       2383     165     13                                 60

       Vertical_Distance_To_Hydrology  Horizontal_Distance_To_Roadways  \
0                                  95                              780
1                                  29                              404
2                                  42                              509
3                                  71                              474
4                                  34                              390
…                                  …                               …
38496                              17                              108
38497                              12                               95
38498                               7                               90
38499                               5                               90
38500                               4                               67

       Hillshade_9am  Hillshade_Noon  Hillshade_3pm  \
0                188             253            199
1                183             211            164
2                212             212            134
3                226             200            102
4                152             188            168
…                …               …              …
38496            240             237            118
38497            240             237            119
38498            236             241            130
38499            230             245            143
38500            231             244            141

       Horizontal_Distance_To_Fire_Points  …  Soil_Type_31  Soil_Type_32  \
0                                    1410  …             0             0
1                                     300  …             0             0
2                                     421  …             0             0
3                                     283  …             0             0
4                                     190  …             0             0
…                                     …    …             …             …
38496                                 837  …             0             0
38497                                 845  …             0             0
38498                                 854  …             0             0
38499                                 864  …             0             0
38500                                 875  …             0             0
```

```
       Soil_Type_33  Soil_Type_34  Soil_Type_35  Soil_Type_36  Soil_Type_37  \
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
…               …             …             …             …             …
38496             0             0             0             0             0
38497             0             0             0             0             0
38498             0             0             0             0             0
38499             0             0             0             0             0
38500             0             0             0             0             0

       Soil_Type_38  Soil_Type_39  Cover_Type
0                 0             0           0
1                 0             0           0
2                 0             0           0
3                 0             0           0
4                 0             0           0
…               …             …           …
38496             0             0           0
38497             0             0           0
38498             0             0           0
38499             0             0           0
38500             0             0           0

[38501 rows x 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
- `Horizontal_Distance_To_Hydrology`: Horizontal dist to nearest surface water features in meters
- `Vertical_Distance_To_Hydrology`: Vertical dist to nearest surface water features in meters
- `Horizontal_Distance_To_Roadways`: Horizontal dist to nearest roadway in meters
- `Hillshade_9am`: Hillshade index at 9am, summer solstice
- `Hillshade_Noon`: Hillshade index at noon, summer solstice
- `Hillshade_3pm`: Hillshade index at 3pm, summer solstice
- `Horizontal_Distance_To_Fire_Points`: Horizontal dist to nearest wildfire ignition points, meters
- `Wilderness_Area_x`: Wilderness area designation (3 columns)
- `Soil_Type_x`: Soil Type designation (39 columns)

- `Cover_Type`: 1 for cottonwood/willow, 0 for ponderosa pine

This is also an imbalanced dataset, since cottonwood/willow trees are relatively rare in this forest:

[4]:
```
# 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.

1.3.2 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.

1.3.3 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 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 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.

1.3.4 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

1.4 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.

[8]:
```
# Your code here
from sklearn.model_selection import train_test_split

y = df["Cover_Type"]
X = df.drop("Cover_Type", axis = 1)

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.

[10]:
```
# Your code here
from sklearn.preprocessing import StandardScaler

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

The following code checks that everything is set up correctly:

[11]:
```
# 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
```

1.5 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.

[13]:
```
# 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 = -1*np.mean(cross_val_score(knn_baseline_model,
                                        X_train_scaled,
                                        y_train,
                                        scoring="neg_log_loss"))

knn_baseline_log_loss
```

[13]:
```
0.1255288892455634
```

[19]:
```
### From MySelf
knn_baseline_model.fit(X_train_scaled, y_train)
print("Training Score: ", knn_baseline_model.score(X_train_scaled, y_train))
print("Test Score :", knn_baseline_model.score(X_test_scaled, y_test))
```

```
Training Score:  0.9897835497835498
Test Score : 0.984209432786204
```

Our best logistic regression model had a log loss of 0.13031294393913376

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

[14]:
```
# Replace None with appropriate text
"""
Our score is 0.1255 which is smaller than 0.1303 so our untuned kNN model
performs better than our tunned Logistic Regression Model. However, the current
model is slower
```

```
"""

# ## From GH

# """
# 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
# """
```

[14]:
```
'\nOur log loss is better with the vanilla (un-tuned) kNN model\nthan it was
with the tuned logistic regression model\n\nIt was also much slower, taking
around a minute to complete\nthe cross-validation on this machine\n\nIt depends
on the business case whether this is really a better\nmodel\n'
```

1.6 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.

[20]:
```
# ## GH
# """
# Your work will not look identical to this, and that is ok!
# The goal is that there should be an explanation for
# everything you try, and accurate reporting on the outcome
# """

# """
# Maybe we are overfitting, since the default neighbors of 5
# seems small compared to the large number of records in this
# dataset. Let's increase that number of neighbors 10x to see
# if it improves the results.
# """

# Your code here (add more cells as needed)

# Creating the model
knn_baseline_model_1 = KNeighborsClassifier(n_neighbors=50)

# Perform cross-validation
knn_baseline_log_loss_1 = -1*np.mean(cross_val_score(knn_baseline_model_1,
```

```
                                        X_train_scaled,
                                        y_train,
                                        scoring="neg_log_loss"))

knn_baseline_log_loss_1
```

[20]: 0.078613760394212

[21]:
```
knn_baseline_model_1.fit(X_train_scaled, y_train)
print("Training Score: ", knn_baseline_model_1.score(X_train_scaled, y_train))
print("Test Score: ", knn_baseline_model_1.score(X_test_scaled, y_test))
```

```
Training Score:  0.9699740259740259
Test Score:  0.9690421774361105
```

[ ]:
```
"""
The Neg-Log-Loss improved much better, but our score compared to the previous
model decreased. But it seems that we do not have overffiting

"""

# ### GH

# """
# Great, that looks good. What if we keep that number of
# neighbors, and change the distance metric from euclidean
# to manhattan?
# """
```

[22]:
```
# Your code here (add more cells as needed)

# Creating the model
knn_baseline_model_2 = KNeighborsClassifier(n_neighbors=50, metric =␣
 ↪"manhattan" )

# Perform cross-validation
knn_baseline_log_loss_2 = -1*np.mean(cross_val_score(knn_baseline_model_2,
                                        X_train_scaled,
                                        y_train,
                                        scoring="neg_log_loss"))

knn_baseline_log_loss_2
```

[22]: 0.07621145166565102

[23]:
```
knn_baseline_model_2.fit(X_train_scaled, y_train)
print("Training Score: ", knn_baseline_model_2.score(X_train_scaled, y_train))
```

```
print("Test Score: ", knn_baseline_model_2.score(X_test_scaled, y_test))
```

```
Training Score:  0.9719480519480519
Test Score:  0.9716393102015375
```

[27]:
```
"""
Log Loss is almost the same but scores are slightly better than previous one
but still our base model is better in score values but not in Log-Loss score
"""

# ### GH

# """
# Ok, slightly better but it's a much smaller difference now.

# Maybe we can get even better performance by increasing the
# number of neighbors again.
# """
```

[27]:
```
"\nOk, slightly better but it's a much smaller difference now.\n\nMaybe we can
get even better performance by increasing the\nnumber of neighbors again.\n"
```

[26]:
```
# Your code here (add more cells as needed)

# Creating the model
knn_baseline_model_3 = KNeighborsClassifier(n_neighbors=75, metric =␣
 ↪"manhattan" )

# Perform cross-validation
knn_baseline_log_loss_3 = -1*np.mean(cross_val_score(knn_baseline_model_3,
                                     X_train_scaled,
                                     y_train,
                                     scoring="neg_log_loss"))

print("Log Loss:", knn_baseline_log_loss_3)

knn_baseline_model_3.fit(X_train_scaled, y_train)
print("Training Score: ", knn_baseline_model_3.score(X_train_scaled, y_train))
print("Test Score: ", knn_baseline_model_3.score(X_test_scaled, y_test))
```

```
Log Loss: 0.08591231255583043
Training Score:  0.9685541125541125
Test Score:  0.9689382921254934
```

[ ]:
```
# ### From GH

# """
```

```
# While this was still better than when n_neighbors was 5
# (the default), it's worse than n_neighbors being 50

# If we were to build more models, we would probably start
# investigating the space between 5 and 50 neighbors to find
# the best number, but for now we'll just stop and say that
# knn_third_model is our best one.
# """
```

1.7 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.)

[29]:
```
# Your code here
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score


# Creating the model
tree_baseline_model = DecisionTreeClassifier(random_state=42)

# Perform cross-validation
tree_baseline_log_loss = -1*np.mean(cross_val_score(tree_baseline_model,
                                    X_train,
                                    y_train,
                                    scoring="neg_log_loss"))

print("Log Loss:", tree_baseline_log_loss)
tree_baseline_model.fit(X_train, y_train)
print("Training Score:", tree_baseline_model.score(X_train, y_train))
print("Test Score:", tree_baseline_model.score(X_test, y_test))
```

```
Log Loss: 0.7045390124149022
Training Score: 1.0
Test Score: 0.9782879700810305
```

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

[ ]:
```
# Replace None with appropriate text
"""
Our Log Loss for this model is much higher than our best kNN model but it is
much faster
"""
```

```
# ### GH
# """
# 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.
# """
```

1.8 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.

[30]:
```
# Your code here (add more cells as needed)
# Creating the model
tree_baseline_model_1 = DecisionTreeClassifier(random_state=42, max_depth = 5)

# Perform cross-validation
tree_baseline_log_loss_1 = -1*np.mean(cross_val_score(tree_baseline_model_1,
                                      X_train,
                                      y_train,
                                      scoring="neg_log_loss"))

print("Log Loss:", tree_baseline_log_loss_1)
tree_baseline_model_1.fit(X_train, y_train)
print("Training Score:", tree_baseline_model_1.score(X_train, y_train))
print("Test Score:", tree_baseline_model_1.score(X_test, y_test))
```

```
Log Loss: 0.11887320104683834
Training Score: 0.9683809523809523
Test Score: 0.9674838977768544
```

[44]:
```
# Your code here (add more cells as needed)
# Your code here (add more cells as needed)
# Creating the model
tree_baseline_model_2 = DecisionTreeClassifier(random_state=42,
                                               max_depth = 5,
                                               min_samples_leaf = 10)

# Perform cross-validation
tree_baseline_log_loss_2 = -1*np.mean(cross_val_score(tree_baseline_model_2,
                                      X_train,
                                      y_train,
                                      scoring="neg_log_loss"))

print("Log Loss:", tree_baseline_log_loss_2)
```

```
tree_baseline_model_2.fit(X_train, y_train)
print("Training Score:", tree_baseline_model_2.score(X_train, y_train))
print("Test Score:", tree_baseline_model_2.score(X_test, y_test))
```

```
Log Loss: 0.11611292651380065
Training Score: 0.9681385281385282
Test Score: 0.9672761271556202
```

[42]:
```
# Your code here (add more cells as needed)
# Your code here (add more cells as needed)
# Creating the model
tree_baseline_model_3 = DecisionTreeClassifier(random_state=42, max_depth = 4)

# Perform cross-validation
tree_baseline_log_loss_3 = -1*np.mean(cross_val_score(tree_baseline_model_3,
                                      X_train,
                                      y_train,
                                      scoring="neg_log_loss"))

print("Log Loss:", tree_baseline_log_loss_3)
tree_baseline_model_3.fit(X_train, y_train)
print("Training Score:", tree_baseline_model_3.score(X_train, y_train))
print("Test Score:", tree_baseline_model_3.score(X_test, y_test))
```

```
Log Loss: 0.13012531033354707
Training Score: 0.9596883116883117
Test Score: 0.9588614169956368
```

[45]:
```
### GH

# """
# There are a lot of ways to reduce overfitting in this
# model, so it can be overwhelming to choose which one
# to try!

# Let's start with increasing min_samples_leaf by an
# order of magnitude. This is conceptually the most
# similar to increasing the number of neighbors, although
# the process for determining similarity of neighbors and
# samples in the same leaf is quite different.
# """

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,
```

```
                                          scoring="neg_log_loss").mean()
dtree_second_log_loss
```

[45]: 0.28719567271672036

[46]:
```
#### GH

# """
# Ok, increasing the minimum samples per leaf from 1 to 10
# helped reduce overfitting. What if we increase them again?
# """

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="neg_log_loss").mean()
dtree_third_log_loss
```

[46]: 0.11894015917756935

[47]:
```
### GH

# """
# Now we are getting scores in the same range as the best
# logistic regression model or the baseline kNN model

# Wait, we just realized that this is a very imbalanced dataset,
# but we haven't told the model that. Let's try using the same
# min_samples_leaf as before, and also specifying the class weight
# """

dtree_fourth_model = DecisionTreeClassifier(random_state=42,
                                            min_samples_leaf=100,
                                            class_weight="balanced")

dtree_fourth_log_loss = -cross_val_score(dtree_fourth_model,
                                         X_train, y_train,
                                         scoring="neg_log_loss").mean()
dtree_fourth_log_loss
```

[47]: 0.20808868577091694

[48]:
```
### GH

# """
```

```
# Oh well, sometimes that backfires when the model overcompensates
# trying to create the right balance. We'll leave off the
# class_weight hyperparameter for now.

# We also notice that this dataset has a lot of dimensions. What if
# we limit the number of features that can be used in a given split,
# while keeping min_samples_leaf the same?
# """

dtree_fifth_model = DecisionTreeClassifier(random_state=42,
                                           min_samples_leaf=100,
                                           max_features="sqrt")

dtree_fifth_log_loss = -cross_val_score(dtree_fifth_model,
                                        X_train, y_train,
                                        scoring="neg_log_loss").mean()
dtree_fifth_log_loss
```

[48]: 0.14254711982373908

[49]:
```
### GH

# """
# Still not better than dtree_third_model

# Let's try one more value for min_samples_leaf
# """

dtree_sixth_model = DecisionTreeClassifier(random_state=42, min_samples_leaf=75)

dtree_sixth_log_loss = -cross_val_score(dtree_sixth_model,
                                        X_train, y_train,
                                        scoring="neg_log_loss").mean()
dtree_sixth_log_loss
```

[49]: 0.11510335541930405

[50]:
```
### From GitHub

# """
# That looks good. Maybe in the future we would do something more
# systematic (like a grid search) but for now we'll say that the
# sixth model is the best one of the decision tree models
# """
```

1.9 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.

[51]:
```
# 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)
```

[51]:
```
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.

[53]:
```
# 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?

[ ]:
```
# 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
# """
```

1.10 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.

2 For My Practice:

Here I am going to work with other Ensemble methods such as `Random Forest` and `XGBoost` and I will search grid to find the best match.

[58]:
```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪log_loss

from xgboost import XGBClassifier

import warnings
warnings.filterwarnings('ignore')
```

2.1 Bagged Tree

[65]:
```
bagged_tree = BaggingClassifier(
                DecisionTreeClassifier(criterion="gini",
                max_depth = 10),
```

```
                n_estimators =50
)

bagged_tree_log_loss = -cross_val_score(bagged_tree,
                                        X_train, y_train,
                                        scoring="neg_log_loss").mean()


bagged_tree.fit(X_train, y_train)

print("Log Loss:", bagged_tree_log_loss)
print("Traninig Score:", bagged_tree.score(X_train, y_train))
print("Test Score:", bagged_tree.score(X_test, y_test))

print()

preds_bagged = bagged_tree.predict(X_test)
probs_bagged = bagged_tree.predict_proba(X_test)

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

```
Log Loss: 0.050236130156247505
Traninig Score: 0.9923116883116884
Test Score: 0.9821317265738625

log loss:  0.05109993543837824
accuracy:  0.9821317265738625
precision: 0.8837555886736215
recall:    0.8631732168850073
```

2.2 Random Forest

[66]:
```
forest = RandomForestClassifier(n_estimators=100, max_depth = 15)

forest_log_loss = -cross_val_score(forest,
                                   X_train, y_train,
                                   scoring="neg_log_loss").mean()



forest.fit(X_train, y_train)

print("Log Loss:", forest_log_loss)
print("Traninig Score:", forest.score(X_train, y_train))
```

```
print("Test Score:", forest.score(X_test, y_test))

print()

preds_forest = forest.predict(X_test)
probs_forest = forest.predict_proba(X_test)

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

```
Log Loss: 0.054744034219359916
Traninig Score: 0.9968138528138528
Test Score: 0.9847288593392894

log loss:  0.05151671945972006
accuracy:  0.9847288593392894
precision: 0.9258675078864353
recall:    0.8544395924308588
```

2.3 XGBoost

[67]:
```
XGB = XGBClassifier()

XGB_log_loss = -cross_val_score(XGB,
                                X_train, y_train,
                                scoring="neg_log_loss").mean()



XGB.fit(X_train, y_train)

print("Log Loss:", XGB_log_loss)
print("Traninig Score:", XGB.score(X_train, y_train))
print("Test Score:", XGB.score(X_test, y_test))

print()

preds_XGB = XGB.predict(X_test)
probs_XGB = XGB.predict_proba(X_test)

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

```
Log Loss: 0.03402142245556726
```

```
Traninig Score: 0.9995844155844156
Test Score: 0.9877415333471847

log loss:  0.03524546260024428
accuracy:  0.9877415333471847
precision: 0.9153284671532846
recall:    0.9126637554585153
```

We can see that untuned `XGBClassifier()` gives the best results, so, now I will choose this model and will try to find a best tuning for this model.

2.4 GridSearchCV with XGBClassifier()

[71]:
```
param_grid = {
'learning_rate': [0.1, 0.2],
'max_depth': [6],
'min_child_weight': [1, 2],
'subsample': [0.5, 0.7],
'n_estimators': [100],
            }

XGB_tunned = XGBClassifier()

grid_clf = GridSearchCV(XGB_tunned, param_grid,
                        scoring='accuracy', n_jobs=1, cv = None)

grid_clf.fit(X_train, y_train)

best_parameters = grid_clf.best_params_

print('Grid Search found the following optimal parameters: ')
for param_name in sorted(best_parameters.keys()):
    print('%s: %r' % (param_name, best_parameters[param_name]))

training_preds = grid_clf.predict(X_train)
test_preds     = grid_clf.predict(X_test)
training_accuracy = accuracy_score(y_train, training_preds)
test_accuracy = accuracy_score(y_test, test_preds)

print('')
print('Training Accuracy: {:.4}%'.format(training_accuracy * 100))
print('Validation accuracy: {:.4}%'.format(test_accuracy * 100))
```

```
Grid Search found the following optimal parameters:
learning_rate: 0.2
max_depth: 6
min_child_weight: 1
n_estimators: 100
```

```
subsample: 0.7

Training Accuracy: 99.81%
Validation accuracy: 98.78%
```

[72]:
```
preds_grid_clf = grid_clf.predict(X_test)
probs_grid_clf = grid_clf.predict_proba(X_test)

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

```
log loss:  0.03600730954330057
accuracy:  0.9878454186578017
precision: 0.9094827586206896
recall:    0.9213973799126638
```

[73]:
```
XGB_tunned = XGBClassifier(learning_rate = 0.2,
                           max_depth = 6,
                           min_child_weight = 1,
                           n_estimators = 100,
                           subsample = 0.7)

XGB_tunned_log_loss = -cross_val_score(XGB_tunned,
                                       X_train, y_train,
                                       scoring="neg_log_loss").mean()



XGB_tunned.fit(X_train, y_train)

print("Log Loss:", XGB_tunned_log_loss)
print("Traninig Score:", XGB_tunned.score(X_train, y_train))
print("Test Score:", XGB_tunned.score(X_test, y_test))

print()

preds_XGB_tunned = XGB_tunned.predict(X_test)
probs_XGB_tunned = XGB_tunned.predict_proba(X_test)

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

```
Log Loss: 0.0355434885308743
Traninig Score: 0.9980952380952381
```

```
Test Score: 0.9878454186578017

log loss:  0.03600730954330057
accuracy:  0.9878454186578017
precision: 0.9094827586206896
recall:    0.9213973799126638
```

[ ]: