CS 289A – Spring 2023 – Homework 5

Colin Skinner, SID [redacted]

1 Honor Code

I did not collaborate with any students. I did refer to ChatGPT frequently.

"I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."

Signed Colin Skinner

Signature __________ Date 3/31/2023

2 Random Forest Motivation

2.1 (a)

$$\mathbb{E}\left[\frac{1}{n}\sum_{i}^{n} Y_i\right] = \frac{1}{n}\sum_{i}^{n} \mathbb{E}\left[Y_i\right]$$

$$\boxed{= \mu}$$

$$\begin{aligned}\mathrm{Var}\left(\frac{1}{n}\sum_{i}^{n} Y_i\right) &= \frac{1}{n^2}\sum_{i}^{n} \mathrm{Var}(Y_i) \\ &= \frac{1}{n^2}\sum_{i}^{n} \sigma^2 \\ &= \boxed{\frac{\sigma^2}{n}}\end{aligned}$$

2.2 (b)

2.2.1 (i)

For sampling with replacement, for any one trial (i.e. picking a sample)

$$P(\text{chosen}) = \frac{1}{n}$$

and

$$P(\text{not chosen}) = 1 - \frac{1}{n}$$

If we create a subsample with n points, then the probability that a particular sample never gets selected is

$$P(\text{never selected}) = \left(1 - \frac{1}{n}\right)^n$$

For very large n

$$\lim_{n\to\infty}\left(1 - \frac{1}{n}\right)^n = \lim_{n\to\infty}\left(1 + \frac{x}{n}\right)^n$$

Where x = -1

$$= e^x$$

$$= e^{-1}$$

$$= 0.3678...$$

Therefore, about 37% of samples never get selected, which also means about 63% get selected at least once.

2.2.2 (ii)

The number of trees to include can be determined the usual way using a range of possible values and cross-validation, choosing the T which gives the optimal performance. Additionally, rather than conventional cross-validation, T can be found by checking the out-of-bag error as the number of trees increases, and continuing to increase T until the out-of-bag error stabilizes.

2.3 (c)

Source: https://www.probabilitycourse.com/chapter6/6_1_2_sums_random_variables.php

$$\mathrm{Var}\left(\frac{1}{n}\sum_{i=1}^{n} Z_i\right) = \frac{1}{n^2}\mathrm{Var}\left(\sum_{i=1}^{n} Z_i\right)$$

$$= \frac{1}{n^2}\left(\sum_{i=1}^{n}\mathrm{Var}(Z_i) + 2\sum_{i<j}\mathrm{Cov}(Z_i, Z_j)\right)$$

$$= \frac{1}{n^2}\left(n\sigma^2 + 2\sum_{i<j}\rho\right)$$

$$= \frac{1}{n^2}\left(n\sigma^2 + 2((n-1)+(n-2)+...+2+1)\rho\right)$$

$$= \frac{1}{n^2}\left(n\sigma^2 + 2\frac{(n-1)((n-1)+1)}{2}\rho\right)$$

$$= \frac{1}{n^2}\left(n\sigma^2 + n(n-1)\rho\right)$$

$$\boxed{= \frac{1}{n}\left(\sigma^2 + (n-1)\rho\right)}$$

3 Gaussian Kernels

3.1 (a)

In this case we only need to consider the diagonal entries of K, $k(x_1, x_1)$ and $k(x_2, x_2)$

$$k(x_1, x_1) = \lim_{\sigma \to 0} \exp\left(-\frac{\|0\|_2^2}{2\sigma^2}\right)$$

$$= \exp(0)$$

$$= 1$$

$$k(x_2, x_2)$$

Thus K is just the identity matrix and the optimization problem becomes

$$a^* = \arg\min_a \|a - y\|_2^2$$

$$= y$$

or

$$\boxed{a^* = \begin{bmatrix} 1 \\ -1 \end{bmatrix}}$$

as this obviously gives a zero magnitude. This gives the classifier

$$\hat{f}(x) = \text{sign}\left(\sum_{i=1}^{n} a_i k(x_i, x)\right)$$

$$= \boxed{\text{sign}\Big(k(1, x) - k(-1, x)\Big)}$$

3.2 (b)

In this case K is a matrix of ones and then we would have

$$K\mathbf{a} = \begin{bmatrix} \sum_i^n a_i \\ \sum_i^n a_i \\ . \\ . \\ . \\ \sum_i^n a_i \end{bmatrix}$$

Let

$$C = \sum_i^n a_i$$

Then

$$\|K\mathbf{a} - \mathbf{y}\|_2^2 = \|C\mathbf{1} - \mathbf{y}\|_2^2$$

$$= (C - y_1)^2 + (C - y_2)^2 + ... + (C - y_n)^2$$

and

$$\frac{\partial}{\partial \mathbf{a}} \left[(C - y_1)^2 + (C - y_2)^2 + ... + (C - y_n)^2\right] = 0$$

$$2\frac{\partial C}{\partial \mathbf{a}} (C - y_1 + C - y_2 + ... + C - y_n) = 0$$

$$nC - (y_1 + y_2 + ... + y_n) = 0$$

$$C = \frac{(y_1 + y_2 + ... + y_n)}{n}$$

In other words, the optimum is when $C = \sum_i^n a_i$ is equal to the average of the labels, and of course if there are equal numbers of $+1$ and -1 labels their average is 0 and we have

$$\sum_i^n a_i = 0$$

In which case

$$\mathbf{a}^* = \mathbf{0}$$

is *an* optimum solution. To show it is a minimum note that by Cauchy-Schwarz

$$\|K\mathbf{a} - \mathbf{y}\|_2 \leq \|K\mathbf{a}\|_2 + \|\mathbf{y}\|_2$$

$$= \sigma_{\max}(K)\|\mathbf{a}\|_2 + \|\mathbf{y}\|_2$$

and since singular values are always non-negative we see that the above is minimized when $\mathbf{a} = \mathbf{a}^* = \mathbf{0}$, and thus $\mathbf{a}^* = \mathbf{0}$ is a minimum.

Q.E.D.

In this case, the classifier is

$$\hat{f}(x) = \text{sign}\Big(0\Big)$$

$$\boxed{\hat{f}(x) = 1}$$

or in other words, every point is considered in class, which makes sense if you increase the bandwidth parameter to infinity because you are saying all points are "close" to one another and you fail to build a functional classifier.

4 Decision Trees for Classification

I was unable to finish, but have submitted my code to show what progress I made.

In [129…

```python
# You may want to install "gprof2dot"
import io
from collections import Counter

import numpy as np
import pandas as pd
import scipy.io
import sklearn.model_selection
import sklearn.tree
from numpy import genfromtxt
from scipy import stats
from sklearn.base import BaseEstimator, ClassifierMixin
from math import log2 as log

import pydot
```

In [319…

```python
eps = 1e-5  # a small number

def entropy(y):

    if len(y) == 0:
        return 0
    #assumes labels are either one or zero
    pc = sum(y)/len(y)
    pd = 1 - pc

    return -(pc*log(pc+eps)+pd*log(pd+eps))
```

In [321…

```python
class DecisionTree:
    def __init__(self, max_depth=3, feature_labels=None):
        self.max_depth = max_depth
        self.features = feature_labels
        self.left, self.right = None, None  # for non-leaf nodes
        self.split_idx, self.thresh = None, None  # for non-leaf nodes
        self.data, self.pred = None, None  # for leaf nodes

    @staticmethod
    def information_gain(X, y, thresh):
        # TODO: implement information gain function

        lc = [y[i] for i in range(len(X)) if X[i]>=thresh]
        rc = [y[i] for i in range(len(X)) if X[i]<thresh]

        hs = entropy(y)
        hafter = (len(lc)*entropy(lc)+len(rc)*entropy(rc))/len(y)

        return hs-hafter

    @staticmethod
    def gini_impurity(X, y, thresh):
        # TODO: implement gini impurity function

        lc = [y[i] for i in range(len(X)) if X[i]>=thresh]
        rc = [y[i] for i in range(len(X)) if X[i]<thresh]

        l_gini = (1-(sum(lc)/len(lc))**2-(1-(sum(lc)/len(lc)))**2)
        r_gini = (1-(sum(rc)/len(rc))**2-(1-(sum(rc)/len(rc)))**2)
```

```
        return l_gini*len(lc)/len(y) + r_gini*len(rc)/len(y)

    def split(self, X, y, idx, thresh):
        X0, idx0, X1, idx1 = self.split_test(X, idx=idx, thresh=thresh)
        y0, y1 = y[idx0], y[idx1]
        return X0, y0, X1, y1

    def split_test(self, X, idx, thresh):
        idx0 = np.where(X[:, idx] < thresh)[0]
        idx1 = np.where(X[:, idx] >= thresh)[0]
        X0, X1 = X[idx0, :], X[idx1, :]
        return X0, idx0, X1, idx1


    def fit(self, X, y):
        if self.max_depth > 0:
            print(self.max_depth)
            # compute entropy gain for all single-dimension splits,
            # thresholding with a linear interpolation of 10 values
            gains = []
            # The following logic prevents thresholding on exactly the minimum
            # or maximum values, which may not lead to any meaningful node
            # splits.
            thresh = np.array([
                np.linspace(np.min(X[:, i]) + eps, np.max(X[:, i]) - eps, num=10)
                for i in range(X.shape[1])
            ])
            for i in range(X.shape[1]):
                #passes the datapoints for a feature, the labels and a threshold value
                #all the gains on all the features if they were added as the next node
                gains.append([self.information_gain(X[:, i], y, t) for t in thresh[i,

            gains = np.nan_to_num(np.array(gains))

            self.split_idx, thresh_idx = np.unravel_index(np.argmax(gains), gains.shap
            self.thresh = thresh[self.split_idx, thresh_idx]
            X0, y0, X1, y1 = self.split(X, y, idx=self.split_idx, thresh=self.thresh)
            if X0.size > 0 and X1.size > 0:
                self.left = DecisionTree(
                    max_depth=self.max_depth - 1, feature_labels=self.features)
                self.left.fit(X0, y0)
                self.right = DecisionTree(
                    max_depth=self.max_depth - 1, feature_labels=self.features)
                self.right.fit(X1, y1)
            else:
                self.max_depth = 0
                self.data, self.labels = X, y
                self.pred = stats.mode(y).mode[0]
        else:
            self.data, self.labels = X, y
            self.pred = stats.mode(y).mode[0]
        return self

    def predict(self, X):
        if self.max_depth == 0:
            return self.pred * np.ones(X.shape[0])
        else:
            X0, idx0, X1, idx1 = self.split_test(X, idx=self.split_idx, thresh=self.th
            yhat = np.zeros(X.shape[0])
```

```python
            yhat[idx0] = self.left.predict(X0)
            yhat[idx1] = self.right.predict(X1)
            return yhat

    def __repr__(self):
        if self.max_depth == 0:
            return "%s (%s)" % (self.pred, self.labels.size)
        else:
            return "[%s < %s: %s | %s]" % (self.features[self.split_idx],
                                             self.thresh, self.left.__repr__(),
                                             self.right.__repr__())
```

In [211…

```python
class BaggedTrees(BaseEstimator, ClassifierMixin):
    def __init__(self, params=None, n=200):
        if params is None:
            params = {}
        self.params = params
        self.n = n
        self.decision_trees = [
            sklearn.tree.DecisionTreeClassifier(random_state=i, **self.params)
            for i in range(self.n)
        ]

    def fit(self, X, y):
        # TODO: implement function
        pass

    def predict(self, X):
        # TODO: implement function
        pass


class RandomForest(BaggedTrees):
    def __init__(self, params=None, n=200, m=1):
        if params is None:
            params = {}
        # TODO: implement function
        pass


class BoostedRandomForest(RandomForest):
    def fit(self, X, y):
        self.w = np.ones(X.shape[0]) / X.shape[0]  # Weights on data
        self.a = np.zeros(self.n)  # Weights on decision trees
        # TODO: implement function
        return self

    def predict(self, X):
        # TODO: implement function
        pass
```

In [4]:

```python
def preprocess(data, fill_mode=True, min_freq=10, onehot_cols=[]):
    # fill_mode = False

    # Temporarily assign -1 to missing data
    data[data == ''] = '-1'

    # Hash the columns (used for handling strings)
    onehot_encoding = []
```

```
    onehot_features = []
    for col in onehot_cols:
        counter = Counter(data[:, col])
        for term in counter.most_common():
            if term[0] == '-1':
                continue
            if term[-1] <= min_freq:
                break
            onehot_features.append(term[0])
            onehot_encoding.append((data[:, col] == term[0]).astype(float))
        data[:, col] = '0'
    onehot_encoding = np.array(onehot_encoding).T
    data = np.hstack([np.array(data, dtype=float), np.array(onehot_encoding)])

    # Replace missing data with the mode value. We use the mode instead of
    # the mean or median because this makes more sense for categorical
    # features such as gender or cabin type, which are not ordered.
    if fill_mode:
        for i in range(data.shape[-1]):
            mode = stats.mode(data[((data[:, i] < -1 - eps) +
                                    (data[:, i] > -1 + eps))][:, i]).mode[0]
            data[(data[:, i] > -1 - eps) * (data[:, i] < -1 + eps)][:, i] = mode

    return data, onehot_features
```

In [6]:
```
def evaluate(clf):
    print("Cross validation", sklearn.model_selection.cross_val_score(clf, X, y))
    if hasattr(clf, "decision_trees"):
        counter = Counter([t.tree_.feature[0] for t in clf.decision_trees])
        first_splits = [(features[term[0]], term[1]) for term in counter.most_common()
        print("First splits", first_splits)
```

In [145…
```
if __name__ == "__main__":
    dataset = "titanic"
    params = {
        "max_depth": 5,
        # "random_state": 6,
        "min_samples_leaf": 10,
    }
    N = 100

    if dataset == "titanic":
        # Load titanic data
        path_train = './dataset/titanic/titanic_training.csv'
        data = genfromtxt(path_train, delimiter=',', dtype=None, encoding=None)
        path_test = './dataset/titanic/titanic_test_data.csv'
        test_data = genfromtxt(path_test, delimiter=',', dtype=None, encoding=None)
        y = data[1:, -1]  # label = survived
        class_names = ["Died", "Survived"]
        labeled_idx = np.where(y != '')[0]

        y = np.array(y[labeled_idx])
        y = y.astype(float).astype(int)


        print("\n\nPart (b): preprocessing the titanic dataset")
        X, onehot_features = preprocess(data[1:, :-1], onehot_cols=[1, 5, 7, 8])
        X = X[labeled_idx, :]
        Z, _ = preprocess(test_data[1:, :], onehot_cols=[1, 5, 7, 8])
```

```
        assert X.shape[1] == Z.shape[1]
        features = list(data[0, :-1]) + onehot_features

    elif dataset == "spam":
        features = [
            "pain", "private", "bank", "money", "drug", "spam", "prescription", "creat
            "height", "featured", "differ", "width", "other", "energy", "business", "r
            "volumes", "revision", "path", "meter", "memo", "planning", "pleased", "re
            "semicolon", "dollar", "sharp", "exclamation", "parenthesis", "square_brac
            "ampersand"
        ]
        assert len(features) == 32

        # Load spam data
        path_train = './dataset/spam/spam_data.mat'
        data = scipy.io.loadmat(path_train)
        X = data['training_data']
        y = np.squeeze(data['training_labels'])
        Z = data['test_data']
        class_names = ["Ham", "Spam"]

    else:
        raise NotImplementedError("Dataset %s not handled" % dataset)

    print("Features:", features)
    print("Train/test size:", X.shape, Z.shape)

    print("\n\nPart 0: constant classifier")
    print("Accuracy", 1 - np.sum(y) / y.size)

    # Basic decision tree
    print("\n\nPart (a-b): simplified decision tree")
    dt = DecisionTree(max_depth=3, feature_labels=features)
    dt.fit(X, y)
    print("Predictions", dt.predict(Z)[:100])

    print("\n\nPart (c): sklearn's decision tree")
    clf = sklearn.tree.DecisionTreeClassifier(random_state=0, **params)
    clf.fit(X, y)
    evaluate(clf)
    out = io.StringIO()

    # You may want to install "gprof2dot"
    sklearn.tree.export_graphviz(
        clf, out_file=out, feature_names=features, class_names=class_names)
    graph = pydot.graph_from_dot_data(out.getvalue())
#     pydot.graph_from_dot_data(out.getvalue())[0].write_pdf("%s-tree.pdf" % dataset)

    # TODO: implement and evaluate!
```

```
Part (b): preprocessing the titanic dataset
Features: ['pclass', 'sex', 'age', 'sibsp', 'parch', 'ticket', 'fare', 'cabin', 'emba
rked', 'male', 'female', 'S', 'C', 'Q']
Train/test size: (999, 14) (310, 14)


Part 0: constant classifier
Accuracy 0.6166166166166166


Part (a-b): simplified decision tree
Predictions [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0.]


Part (c): sklearn's decision tree
Cross validation [0.795      0.825      0.805      0.755      0.74371859]
```

Titanic preprocessing

In [330…

```
#import the Titanic training data

titanic_train = pd.read_csv('dataset/titanic/titanic_training.csv')
```

In [331…

```
titanic_train
```

Out[331]:

	pclass	sex	age	sibsp	parch	ticket	fare	cabin	embarked	survived
0	1.0	female	40.0	1.0	1.0	16966	134.5000	E34	C	1.0
1	3.0	male	33.0	0.0	0.0	345780	9.5000	NaN	S	0.0
2	3.0	male	3.0	4.0	2.0	347077	31.3875	NaN	S	1.0
3	2.0	female	50.0	0.0	1.0	230433	26.0000	NaN	S	1.0
4	3.0	female	16.0	1.0	1.0	2625	8.5167	NaN	C	1.0
...	...	...	...	...	...	...	...	...	...	...
995	1.0	male	54.0	0.0	0.0	17463	51.8625	E46	S	0.0
996	3.0	female	NaN	3.0	1.0	4133	25.4667	NaN	S	0.0
997	3.0	male	18.0	1.0	0.0	3101267	6.4958	NaN	S	0.0
998	2.0	male	31.0	0.0	0.0	244270	13.0000	NaN	S	1.0
999	3.0	female	24.0	0.0	2.0	PP 9549	16.7000	G6	S	1.0

1000 rows × 10 columns

In [332…

```
#get the number and percentage of missing data points for each column

nulls = pd.DataFrame(columns=['feature', 'n null', 'percent null'])
```

```
count = 0
for col in titanic_train.columns:
    row = {'feature': col, 'n null':titanic_train[col].isnull().sum(),
           'percent null':titanic_train[col].isnull().sum()/len(titanic_train)}
    nulls.loc[count] = row
    count+=1
```

In [333…

```
nulls
```

Out[333]:

	feature	n null	percent null
0	pclass	1	0.001
1	sex	1	0.001
2	age	205	0.205
3	sibsp	1	0.001
4	parch	1	0.001
5	ticket	1	0.001
6	fare	2	0.002
7	cabin	774	0.774
8	embarked	3	0.003
9	survived	1	0.001

The vast majority of the datapoint are missing values for the cabin feature so in this case, rather than impute values it make more sense to drop it as a feature. The rest of the features can be kept and the missing values imputed.

In [334…

```
titanic_train = titanic_train.drop('cabin', axis=1)
```

Before imputing the categorical data should be converted to numerical data. We only need to do this for 'sex', 'ticket' and 'embarked'. 'sex' is easy: we can do 0 for male and 1 for female

In [335…

```
#create a copy for preprocessing
titanic_proc = titanic_train.copy()
```

In [336…

```
count = 0
for val in titanic_proc['sex']:
    if val == 'male':
        titanic_proc.loc[count,'sex'] = 0
        count+=1
    else:
        titanic_proc.loc[count,'sex'] = 1
        count+=1
```

Next, I convert the ticket numbers to ints by converting any letters into their ASCII code

In [337…

```
count=0
```

```
for string in titanic_proc['ticket']:

    if isinstance(string, float):
        titanic_proc.loc[count,'ticket'] = int(new_string)
        count+=1

    else:
        new_string = ""

        for char in string:
            if char.isdigit():
                new_string += char
            else:
                new_string += str(ord(char))

        titanic_proc.loc[count,'ticket'] = int(new_string)
        count += 1
```

Next, convert 'embarked' the following way: C=0, Q=1, S=2

In [338…

```
count=0
for val in titanic_proc['embarked']:

    if val == 'C':
        titanic_proc.loc[count,'embarked'] = 0
        count+=1
    elif val == 'Q':
        titanic_proc.loc[count,'embarked'] = 1
        count+=1
    else:
        titanic_proc.loc[count,'embarked'] = 2
        count+=1
```

In [339…

```
titanic_proc
```

Out[339]:

	pclass	sex	age	sibsp	parch	ticket	fare	embarked	survived
0	1.0	1	40.0	1.0	1.0	16966	134.5000	0	1.0
1	3.0	0	33.0	0.0	0.0	345780	9.5000	2	0.0
2	3.0	0	3.0	4.0	2.0	347077	31.3875	2	1.0
3	2.0	1	50.0	0.0	1.0	230433	26.0000	2	1.0
4	3.0	1	16.0	1.0	1.0	2625	8.5167	0	1.0
...	...	...	...	...	...	...	...	..	...
995	1.0	0	54.0	0.0	0.0	17463	51.8625	2	0.0
996	3.0	1	NaN	3.0	1.0	4133	25.4667	2	0.0
997	3.0	0	18.0	1.0	0.0	3101267	6.4958	2	0.0
998	2.0	0	31.0	0.0	0.0	244270	13.0000	2	1.0
999	3.0	1	24.0	0.0	2.0	8080329549	16.7000	2	1.0

1000 rows × 9 columns

In [340…
```
from sklearn.impute import KNNImputer
```

In [341…
```
imputer = KNNImputer(n_neighbors=10)
```

In [342…
```
titanic_imputed = pd.DataFrame(imputer.fit_transform(titanic_proc), columns=titanic_pr
```

In [343…
```
titanic_labels=np.array(titanic_imputed['survived'])
```

In [344…
```
titanic_t_data = np.array(titanic_imputed.drop('survived', axis=1))
```

In [327…
```
titanic_t_data.shape
```

Out[327]:
```
(1000, 8)
```

In [345…
```
classifier = DecisionTree()
```

In [346…
```
classifier.fit(titanic_t_data,titanic_labels)
```

```
3
2
1
1
2
1
1
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\Anaconda3\lib\site-packages\IPython\core\formatters.py in __call__(self, obj)
    700                 type_pprinters=self.type_printers,
    701                 deferred_pprinters=self.deferred_printers)
--> 702             printer.pretty(obj)
    703             printer.flush()
    704             return stream.getvalue()

~\Anaconda3\lib\site-packages\IPython\lib\pretty.py in pretty(self, obj)
    392                         if cls is not object \
    393                                 and callable(cls.__dict__.get('__repr__')):
--> 394                             return _repr_pprint(obj, self, cycle)
    395 
    396             return _default_pprint(obj, self, cycle)

~\Anaconda3\lib\site-packages\IPython\lib\pretty.py in _repr_pprint(obj, p, cycle)
    698     """A pprint that just redirects to the normal repr function."""
    699     # Find newlines and replace them with p.break_()
--> 700     output = repr(obj)
    701     lines = output.splitlines()
    702     with p.group():

~\AppData\Local\Temp\ipykernel_47288\3586919021.py in __repr__(self)
     96             return "%s (%s)" % (self.pred, self.labels.size)
     97         else:
---> 98             return "[%s < %s: %s | %s]" % (self.features[self.split_idx],
     99                                            self.thresh, self.left.__repr__(),
    100                                            self.right.__repr__())

TypeError: 'NoneType' object is not subscriptable
```

In [347…

```
train, one_hots = preprocess(np.array(titanic_train)[:,:-1],  onehot_cols=[1,5,7])
```

In [351…

```
classifier.fit(train,titanic_labels)
```

```
3
```

Out[351]:

```
0.0 (1000)
```

In [ ]: