دانشکده سامانه‌های هوشمند

یادگیری ماشین

استاد درس: دکتر سامان هراتی زاده

تمرین شماره 1

تاریخ ثبت تمرین در سامانه: ...

نام و نام خانوادگی	محمدحسین مازندرانیان فرد
شماره دانشجویی	830402066
شماره تمرین	1
تاریخ ارسال گزارش	1403/08/04
مدت زمان صرف شده برای پاسخ‌دهی به تمرین	10 ساعت

سوال 1) پیاده سازی مدل KNN بر روی دیتاست iris

پیش پردازش:

ابتدا ستون های عددی را با استفاده از روش min-max scaler بین بازه ی صفر تا یک نرمال میکنیم. دلیل اینکار این است.

```preprocessing
scaler = MinMaxScaler()
iris_scaled_data = iris_orig_data.copy()
cols_to_scale = ['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']
iris_scaled_data[cols_to_scale] =
scaler.fit_transform(iris_scaled_data[cols_to_scale])
```

ستون Spices که ستون هدف ما هست را با استفاده از Label Encoder از داده های عددی به categorial تبدیل میکنیم:

```preprocessing
le = LabelEncoder()
iris_scaled_data['Species'] = le.fit_transform(iris_scaled_data['Species'])
```

در ادامه تابعی برای اضافه کردن نویز با مقادیر مختلف تعریف کردیم:

```preprocessing
def add_noise(X_train, y_train, noise_type, noise_level):
    X_train_noisy = X_train.copy()
    y_train_noisy = y_train.copy()

    if noise_type == 'feature':
        noise = np.random.rand(*X_train.shape) * noise_level / 100
        X_train_noisy = X_train_noisy + noise

    elif noise_type == 'label':
        num_samples_to_change = int(len(y_train) * noise_level / 100)
        random_indices = np.random.choice(len(y_train), num_samples_to_change, replace=False)
        unique_labels = y_train.unique()
        for index in random_indices:
            new_label = np.random.choice(unique_labels[unique_labels != y_train.iloc[index]])
            y_train_noisy.iloc[index] = new_label

    return X_train_noisy, y_train_noisy

X_train_noisy20_feature, y_train_noisy20_feature = add_noise(X_train, y_train, 'feature', 20)
X_train_noisy20_label, y_train_noisy20_label = add_noise(X_train, y_train, 'label', 20)

X_train_noisy40_feature, y_train_noisy40_feature = add_noise(X_train, y_train, 'feature', 40)
X_train_noisy40_label, y_train_noisy40_label = add_noise(X_train, y_train, 'label', 40)
```

پیاده سازی مدل KNN:

KNN Python

```python
class KNN:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def euclidean_distance(self, x1, x2):
        return np.sqrt(np.sum((x1 - x2) ** 2))

    def predict(self, X):
        y_pred = [self._predict(x) for x in X]
        return np.array(y_pred)

    def _predict(self, x):
        distances = [self.euclidean_distance(x, x_train) for x_train in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = np.bincount(k_nearest_labels).argmax()
        return most_common
```

و در نهایت با استفاده از Five Fold Validation با مقادیر مختلف k مدل را تست می کنیم:

Five Fold Validation Python

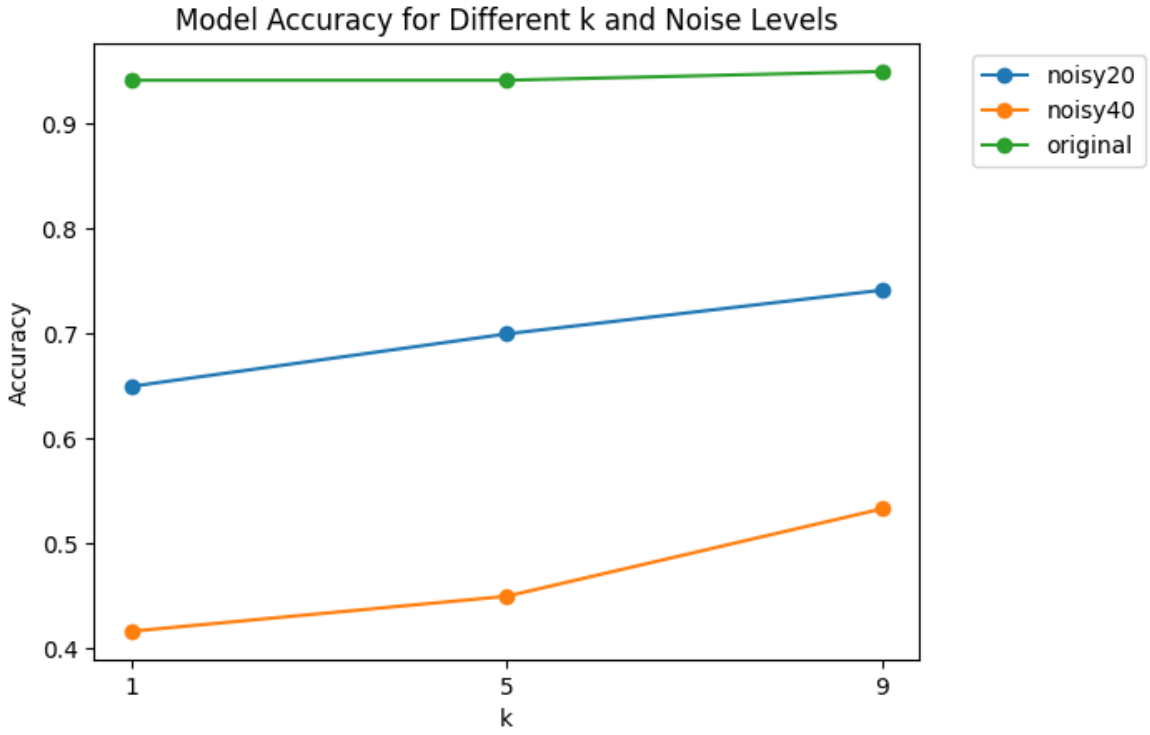
```python
def five_fold_cross_validation(X, y, k_neighbors=3):
    num_folds = 5
    fold_size = len(X) // num_folds
    accuracy_scores = []

    for fold in range(num_folds):
        X_train = np.concatenate([X[:fold * fold_size], X[(fold + 1) * fold_size:]])
        y_train = np.concatenate([y[:fold * fold_size], y[(fold + 1) * fold_size:]])
        X_val = X[fold * fold_size:(fold + 1) * fold_size]
        y_val = y[fold * fold_size:(fold + 1) * fold_size]

        knn = KNN(k=k_neighbors)
        knn.fit(X_train, y_train)

        y_pred = knn.predict(X_val)

        accuracy = np.sum(y_pred == y_val) / len(y_val)
        accuracy_scores.append(accuracy)

    return accuracy_scores

k_values = [1, 5, 9]
noise_levels = ['original', 'noisy20', 'noisy40']
results = []

for k in k_values:
    for noise_level in noise_levels:
        if noise_level == 'original':
            X_train_data = X_train.to_numpy()
            y_train_data = y_train.to_numpy()
        elif noise_level == 'noisy20':
            X_train_data = X_train_noisy20_feature.to_numpy()
            y_train_data = y_train_noisy20_label.to_numpy()
        elif noise_level == 'noisy40':
            X_train_data = X_train_noisy40_feature.to_numpy()
            y_train_data = y_train_noisy40_label.to_numpy()

        accuracy_scores = five_fold_cross_validation(X_train_data, y_train_data,
k_neighbors=k)

        mean_accuracy = np.mean(accuracy_scores)
        results.append([k, noise_level, mean_accuracy])
```

در نهایت دقت(accuracy) مدل روی دیتاهایی مختلف رو در زیر مشاهده می کنیم:

Result Python

```
[[1, 'original', 0.9416666666666667],
 [1, 'noisy20', 0.65],
 [1, 'noisy40', 0.41666666666666663],
 [5, 'original', 0.9416666666666668],
 [5, 'noisy20', 0.7],
 [5, 'noisy40', 0.45],
 [9, 'original', 0.95],
 [9, 'noisy20', 0.7416666666666667],
 [9, 'noisy40', 0.5333333333333333]]
```

در حالت $k = 9$ و روی دیتای اورجینال بیشترین دقت بدست آمده است.



سوال 2) پیاده سازی مدل Prism بر روی دیتاست mushrooms

در پیش پردازش داده ها ردیف هایی که شامل missing value هستند را حذف کردیم، یک راه دیگر این است که missing value ها را به عنوان یک مقدار در نظر بگیریم.

preprocessing — Python

```python
mushroom_orig_data = pd.read_csv('dataset/mushrooms.csv')

data = mushroom_orig_data.copy()
data = data.dropna()

X = data.drop(columns='class')
y = data['class']
```

کلاس PrismClassifier شامل سه متد اصلی fit، predict و rules است که به طور خلاصه هر کدام را در ادامه توضیح می دهیم:

در متد fit ابتدا احتمال مقادیر هر feature مربوط به هر class را محاسبه می کنیم و سپس مشداری که بیشترین تکرار را دارد را انتخاب میکنیم. سپس ردیف هایی که این Rule پوشش می دهد را حذف میکنیم و دوباره روی ردیف های باقی مانده این عمل را تکرار می کنیم تا rules نهایی برای هر کلاس را پیدا کنیم.

prism - fit function — Python

```python
def fit(self, X, y):
    X_copy = X.reset_index(drop=True)
    y_copy = y.reset_index(drop=True)

    for class_label in y_copy.unique():
        perfect_rule_found = False
        while not perfect_rule_found:
            best_rule = None
            best_accuracy = 0

            for feature in X_copy.columns:
                for value in X_copy[feature].unique():
                    rule = (feature, value)
                    covered_indices = X_copy[X_copy[feature] == value].index
                    support = len(covered_indices) / len(X_copy)

                    covered_y = y_copy[X_copy.index.isin(covered_indices)]

                    acc = covered_y.value_counts(normalize=True).get(class_label, 0)

                    if support >= self.min_support and acc > best_accuracy:
                        best_accuracy = acc
                        best_rule = rule

            if best_rule:
                self.final_rules.append((best_rule, class_label))
                X_copy = X_copy[X_copy[best_rule[0]] != best_rule[1]]
                y_copy = y_copy[X_copy.index]
                X_copy = X_copy.reset_index(drop=True)
                y_copy = y_copy.reset_index(drop=True)

                if len(X_copy) == 0:
                    perfect_rule_found = True
            else:
                perfect_rule_found = True
```

در متد predict رو داده های هدف پیمایش انجام می دهیم و بر اساس final rules ای در مرحله ی fit پیدا کرده ایم برای هر ردیف مقدار ستون هدف را محاسبه می کنیم. در این مرحله اگر نتوانستیم مقداری پیدا کنیم unknown را انتخاب می کنیم.

prism - predict function — Python

```python
def predict(self, X):
    pred = []
    for _, instance in X.iterrows():
        predicted_class = self._classify_instance(instance)
        pred.append(predicted_class)
    return np.array(pred)

def _classify_instance(self, instance):
    for rule, class_label in self.final_rules:
        if isinstance(instance[rule[0]], str):
            if instance[rule[0]] == rule[1]:
                return class_label
        else:
            if instance[rule[0]] == rule[1]:
                return class_label

    return 'unknown'
```

در متد rules خروجی rules ها برای هر کلاس p و e را چاپ می کنیم:

prism - rules function — Python

```python
def rules(self):
    class_1_rules = []
    class_0_rules = []
    for rule, class_label in self.final_rules:
        conditions = []
        for f in rule:
            if f != rule[1]:
                conditions.append(f"{f}={rule[1]}")
        rule_string = " AND ".join(conditions)

        if class_label == 'p':
            class_1_rules.append(rule_string)
        else:
            class_0_rules.append(rule_string)
    print("IF", " AND " .join(class_1_rules) + " THEN class=p")

    print("IF", " AND " .join(class_0_rules) + " THEN class=e")
```

نکته ای که در مرحله ی fit وجود دارد مقدار min_support هست که rule هایی که کمتر از این مقدار حست را حساب نمی کنیم. بیشترین مقدار دقت این مدل برای وقتی بود که min_support برابر 0.001 در نظر گرفته شد.

سوال 3) پیاده سازی مدل ID3 بر روی دیتاست titanic

برای پیش پردازش ستون fare را به چهار bin تقسیم کردیم و ستون های sex و embarked هم با استفاده از label encoder به داده های categorial تبدیل شدند:

preprocessing Python

```
data = pd.read_csv('https://raw.githubusercontent.com/modos/ML-
AI/refs/heads/main/datasets/ml_hw_1/titanic.csv')
data=  data.drop(data.columns[0], axis=1)
data["age"] = data["age"].fillna(data["age"].mean())
data["embarked"] = data["embarked"].fillna(data["embarked"].mode()[0])


data['fare_bin'] = pd.qcut(data['fare'], 4, labels=False)
data = data.drop('fare', axis=1)
label_encoder = LabelEncoder()
data["sex"] = label_encoder.fit_transform(data["sex"])
data["embarked"] = label_encoder.fit_transform(data["embarked"])

X = data.drop("survived", axis=1)
y = data["survived"]
```

درخت را با استفاده از معیار های information gain و information gain و gini index و همچنین مقدار عمق درخت train کردیم و با استفاده از از دیتای ولیدیشن به بهترین پارامتر رسیدیم:

best parameters Python

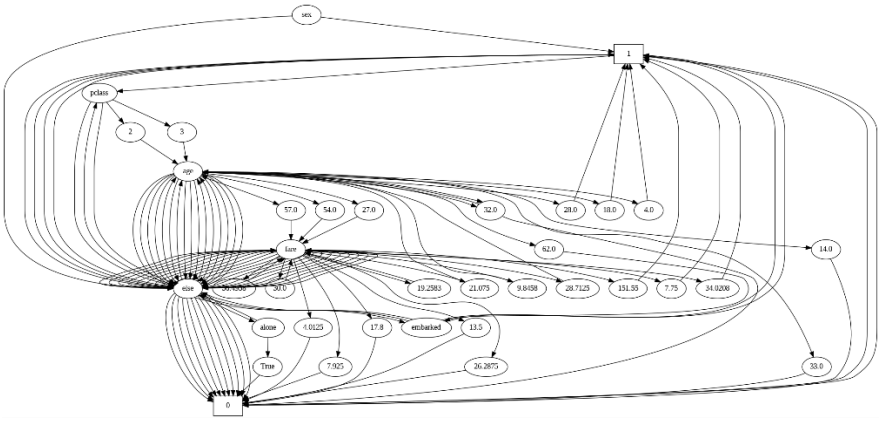
```
Best parameters: {'max_depth': 9, 'min_samples_split': 10, 'criterion':
'gini_index'}
```

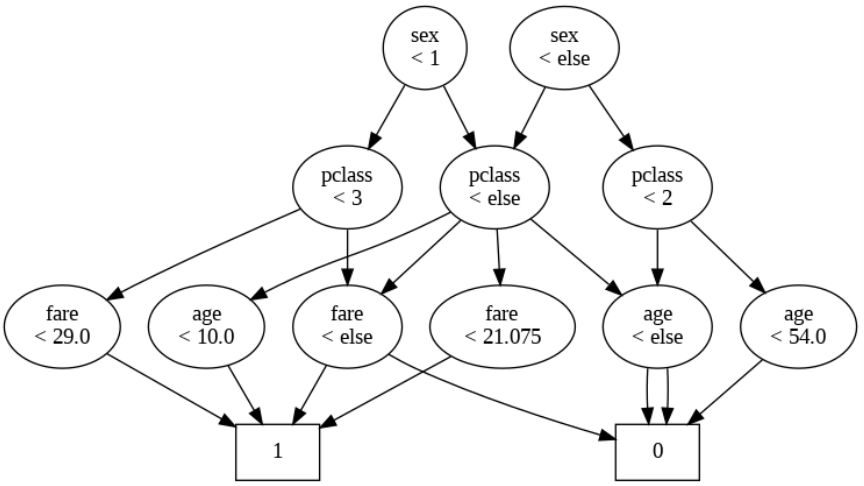
سپس با استفاده از همین معیار ها درخت را روی داده های تست predict کردیم که بهترین دقت به شرح زیر است:

best parameters Python

```
print(best_test_accuracy)
print(best_test_f1)
#0.8181818181818182
#0.74
```

نتایج رسم درخت با استفاده از کتابخانه ی graphviz:





توضیح کلی

- **کلاس ID3DecisionTree**: ویژگی‌ها و معیارهای مختلف برای الگوریتم ID3 را به عنوان پارامتر می‌گیرد و سپس مدل را روی داده‌ها برازش داده و بر اساس ویژگی‌ها و آستانه‌های انتخاب‌شده یک درخت تصمیم‌گیری می‌سازد.
- **متدهای fit و predict**: برای ساخت درخت تصمیم با داده‌های آموزشی و predictبرای پیش‌بینی با داده‌های جدید استفاده می‌شوند.

توضیح متد build_tree

متد build_tree یکی از اجزای اصلی این کلاس است که ساختار درخت تصمیم را به صورت بازگشتی ایجاد می‌کند:

1. **پایان بازگشتی** :این تابع اگر به شرط‌های پایان بازگشتی برسد (یعنی عمق درخت به حداکثر max_depth برسد، تمام نمونه‌ها از یک کلاس باشند یا تعداد نمونه‌ها کمتر از min_samples_split باشد)، یک گره برگ ایجاد می‌کند که دارای مقدار پرتکرارترین کلاس است.

2. **یافتن بهترین ویژگی و آستانه** :اگر به شرایط پایان نرسد، متد find_best_split را فراخوانی می‌کند تا بهترین ویژگی و آستانه‌ای که اطلاعات بیشتری را ارائه می‌دهد، پیدا کند. این انتخاب بهترین ویژگی و آستانه بستگی به معیار انتخاب‌شده مانند اطلاعات متقابل یا جینی دارد.

3. **تقسیم داده‌ها** :اگر ویژگی مناسبی یافت شد، نمونه‌ها را به دو گروه تقسیم می‌کند: یکی برای مقادیر کمتر از آستانه و دیگری برای مقادیر برابر یا بیشتر از آستانه.

4. **ساخت زیر درخت‌ها** :سپس این متد به صورت بازگشتی زیر درخت‌های سمت چپ و راست را می‌سازد و این تقسیمات را ادامه می‌دهد تا به یکی از شرایط پایان برسد.

5. **برگشت ساختار درخت** :پس از ساخت زیر درخت‌های چپ و راست، یک دیکشنری برمی‌گرداند که ویژگی انتخابی را با دو مقدار آستانه و "else" (برای شاخه دیگر) به عنوان زیر درخت‌ها نشان می‌دهد.

این ساختار درخت در نهایت به عنوان مدل ساخته‌شده در متغیر self.tree ذخیره می‌شود و برای پیش‌بینی روی داده‌های جدید استفاده می‌شود.

build tree function — Python

```python
    def _build_tree(self, X, y, depth):
        if len(y) == 0:
            return None
        if depth == self.max_depth or len(set(y)) == 1 or len(X) <
self.min_samples_split:
            return np.bincount(y).argmax()

        best_feature, best_threshold = self._find_best_split(X, y)

        if best_feature is None:
            return np.bincount(y).argmax()

        left_index = X[best_feature] < best_threshold
        right_index = X[best_feature] >= best_threshold

        left_subtree = self._build_tree(X[left_index], y[left_index], depth + 1)
        right_subtree = self._build_tree(X[right_index], y[right_index], depth + 1)

        return {best_feature: {best_threshold: left_subtree, "else":
right_subtree}}
```