

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет ПИиКТ

Системы искусственного интеллекта

Лабораторная работа № 4

Выполнил студент

Набокова Алиса Владиславовна

Группа № Р3320

Преподаватель: Королёва Юлия Александровна

г. Санкт-Петербург

2025

Задание:

1. Реализовать класс DecisionTree.

В классе **должны** быть методы

```
'''python
def __init__(..., classification:bool=true, ...):
    # some code

def predict(...):
    # some code

def fit(...):
    # some code

'''
```

2. Реализовать класс RandomForest.

В классе **должны** быть методы

```
'''python
def __init__(..., classification:bool=true, ...):
    # some code

def predict(...):
    # some code

def fit(...):
    # some code

'''
```

3. Реализовать класс GradientBoosting.

В классе **должны** быть методы

```
'''python
def __init__(..., classification:bool=true, ...):
    # some code
```

```
def predict(...):  
    # some code  
  
def fit(...):  
    # some code  
  
...
```

4. Обучить модели каждого алгоритма на следующих датасетах (проводим мини-соревнование между ними):

регрессия:

<https://www.kaggle.com/datasets/hmavrodiev/london-bike-sharing-dataset>

классификация:

<https://www.kaggle.com/datasets/pankrzysiu/cifar10-python>

5. Объяснить, почему алгоритм А "победил", а почему алгоритм В "проиграл".

На листингах 1-3 представлены классы, реализующие алгоритмы Decision Tree, Random Forest и Gradient Boosting соответственно

Листинг 1. Класс модели Decision Tree

```
class DecisionTree:  
  
    def __init__(self, max_depth=None, min_samples_split=2,  
classification=True):  
  
        self.max_depth = max_depth  
        self.min_samples_split = min_samples_split  
        self.classification = classification  
        self.tree = None
```

```

def _gini(self, y):

    if len(y) == 0:

        return 0.0

    counts = torch.bincount(y.long())

    probs = counts.float() / len(y)

    return 1 - torch.sum(probs ** 2)


def _mse(self, y):

    if len(y) == 0:

        return 0.0

    return torch.var(y.float()) * len(y)


def _best_split(self, X, y):

    best_gain = 0.0

    best_idx = best_thr = None

    n_samples, n_features = X.shape

    parent_imp = self._gini(y) if self.classification else
self._mse(y)

    for idx in range(n_features):

        values = X[:, idx]

        thresholds = torch.unique(values)

        if len(thresholds) > 20:

            step = max(1, len(thresholds) // 15)

            thresholds = thresholds[::-step][:-15]

```

```

        for thr in thresholds:

            left_mask = values <= thr
            right_mask = ~left_mask

            n_left = left_mask.sum().item()
            n_right = (len(y) - n_left)

            if n_left < self.min_samples_split or n_right <
self.min_samples_split:
                continue

            if self.classification:

                gain = parent_imp - (
                    n_left / n_samples * self._gini(y[left_mask]) +
                    n_right / n_samples * self._gini(y[right_mask])
                )

            else:

                gain = parent_imp - (
                    n_left / n_samples * self._mse(y[left_mask]) +
                    n_right / n_samples * self._mse(y[right_mask])
                )

            if gain > best_gain:

                best_gain = gain
                best_idx = idx
                best_thr = thr.item()

    return best_idx, best_thr

```

```

def _build_tree(self, X, y, depth=0):

    n_samples = len(y)

    if (self.max_depth is not None and depth >= self.max_depth) or \
    \
        n_samples < self.min_samples_split or \
        len(torch.unique(y)) == 1:

        value = torch.bincount(y.long()).argmax().item() if
self.classification else y.float().mean().item()

        return {"leaf": True, "value": value}

    idx, thr = self._best_split(X, y)

    if idx is None:

        value = torch.bincount(y.long()).argmax().item() if
self.classification else y.float().mean().item()

        return {"leaf": True, "value": value}

    left_mask = X[:, idx] <= thr

    left = self._build_tree(X[left_mask], y[left_mask], depth +
1)

    right = self._build_tree(X[~left_mask], y[~left_mask], depth +
1)

    return {
        "leaf": False,
        "idx": int(idx),

```

```

        "thr": float(thr),
        "left": left,
        "right": right
    }

def fit(self, X, y):
    X = torch.tensor(X, dtype=torch.float32) if not isinstance(X, torch.Tensor) else X
    y = torch.tensor(y, dtype=torch.long if self.classification else torch.float32)
    self.tree = self._build_tree(X, y)

def _predict_one(self, x, node):
    if node["leaf"]:
        return node["value"]
    if x[node["idx"]] <= node["thr"]:
        return self._predict_one(x, node["left"])
    else:
        return self._predict_one(x, node["right"])

def predict(self, X):
    X = torch.tensor(X, dtype=torch.float32) if not isinstance(X, torch.Tensor) else X
    preds = [self._predict_one(X[i], self.tree) for i in range(X.shape[0])]
    return torch.tensor(preds, dtype=torch.float32 if not self.classification else torch.long)

```

Листинг 2. Класс модели Random Forest

```
class RandomForest:

    def __init__(self, n_estimators=50, max_depth=10,
max_features="sqrt", classification=True):

        self.n_estimators = n_estimators

        self.max_depth = max_depth

        self.max_features = max_features

        self.classification = classification

        self.trees = []


    def fit(self, X, y):

        X = torch.tensor(X, dtype=torch.float32)

        y = torch.tensor(y, dtype=torch.long if self.classification
else torch.float32)

        n, f = X.shape

        max_f = int(f**0.5) if self.max_features == "sqrt" else f


        self.trees = []

        for _ in range(self.n_estimators):

            idx = torch.randint(0, n, (n,))

            X_b = X[idx]

            y_b = y[idx]

            feat_idx = torch.randperm(f)[:max_f]

            X_sub = X_b[:, feat_idx]

            tree = DecisionTree(max_depth=self.max_depth,
classification=self.classification)
```

```
        tree.fit(X_sub, y_b)

        tree.feat_idx = feat_idx

        self.trees.append(tree)

    def predict(self, X):
        X = torch.tensor(X, dtype=torch.float32)
        preds = []
        for tree in self.trees:
            X_sub = X[:, tree.feat_idx]
            preds.append(tree.predict(X_sub))
        preds = torch.stack(preds)

        if self.classification:
            return torch.mode(preds, dim=0).values
        else:
            return preds.float().mean(dim=0)
```

Листинг 3. Класс модели Gradient Boosting

```
class GradientBoosting:

    def __init__(self, n_estimators=50, learning_rate=0.1, max_depth=3,
classification=False, checkpoint_path=None,
                verbose=True):

        self.n_estimators = n_estimators
        self.learning_rate = learning_rate
        self.max_depth = max_depth
        self.classification = classification
        self.checkpoint_path = checkpoint_path
        self.verbose = verbose

        self.models = []
        self.n_classes = None


    def fit(self, X, y):
        X = torch.tensor(X, dtype=torch.float32) if not isinstance(X,
torch.Tensor) else X
        y = torch.tensor(
            y,
            dtype=torch.long if self.classification else torch.float32
        ) if not isinstance(y, torch.Tensor) else y

        if self.classification:
            self.n_classes = len(torch.unique(y))
            self.models = []
            for cls in range(self.n_classes):
```

```

y_bin = (y == cls).float()

self.models.append(self._fit_single(X, y_bin,
cls_id=cls))

else:

    self.models = self._fit_single(X, y, cls_id=None)

def _fit_single(self, X, y, cls_id=None):
    models = []
    raw_pred = torch.zeros(len(y))
    start_iter = 0

    if      self.checkpoint_path      and
os.path.exists(self._ckpt_name(cls_id)):

        ckpt    = torch.load(self._ckpt_name(cls_id),
weights_only=False)

        models = ckpt["models"]
        raw_pred = ckpt["raw_pred"]
        start_iter = len(models)

    if self.verbose:
        print(f"Resume from iteration {start_iter}")

    for i in range(start_iter, self.n_estimators):
        tree    = DecisionTree(max_depth=self.max_depth,
classification=False)

        residual = y - raw_pred
        tree.fit(X, residual)

        update = tree.predict(X)

```

```

        raw_pred += self.learning_rate * update

        models.append(tree)

    if self.verbose:

        print(f"[{i+1}/{self.n_estimators}]")

    if self.checkpoint_path:

        os.makedirs(os.path.dirname(self.checkpoint_path),
exist_ok=True)

        torch.save(
            {
                "models": models,
                "raw_pred": raw_pred
            },
            self._ckpt_name(cls_id)
        )

    return models

def _ckpt_name(self, cls_id):

    if cls_id is None:

        return self.checkpoint_path

    return f"{self.checkpoint_path}_{cls_id}.pt"

def predict(self, X):

    X = torch.tensor(X, dtype=torch.float32) if not isinstance(X,
torch.Tensor) else X

```

```

    if self.classification:

        scores = []

        for cls_models in self.models:

            raw_pred = torch.zeros(X.shape[0])

            for tree in cls_models:

                raw_pred += self.learning_rate * tree.predict(X)

            scores.append(raw_pred)

    return torch.argmax(torch.stack(scores), dim=0)

else:

    raw_pred = torch.zeros(X.shape[0])

    for tree in self.models:

        raw_pred += self.learning_rate * tree.predict(X)

    return raw_pred

```

На следующих листингах 4-5 представлена предобработка данных для регрессии а также код обучения моделей, вывод метрик и графиков

Листинг 4. Предобработка данных для регрессии

```

df_bikes['timestamp'] = pd.to_datetime(df_bikes['timestamp'])

df_bikes['hour'] = df_bikes['timestamp'].dt.hour

df_bikes['day'] = df_bikes['timestamp'].dt.day

df_bikes['month'] = df_bikes['timestamp'].dt.month

df_bikes['weekday'] = df_bikes['timestamp'].dt.weekday

features = ['t1', 't2', 'hum', 'wind_speed', 'weather_code',
'is_holiday', 'is_weekend', 'season', 'hour', 'month', 'weekday']

target = 'cnt'

```

```

df_bikes = df_bikes.sort_values('timestamp')

n = len(df_bikes)

train_df = df_bikes.iloc[:int(0.8 * n)]

test_df = df_bikes.iloc[int(0.8 * n):]

X_reg_train = train_df[features].values

y_reg_train = train_df[target].values

X_reg_test = test_df[features].values

y_reg_test = test_df[target].values

```

Листинг 5. Код обучения моделей для регрессии, счет метрик и графиков

```

models_reg = {

    "DecisionTree": DecisionTree(max_depth=12, min_samples_split=3,
classification=False),

    "RandomForest": RandomForest(n_estimators=70, max_depth=12,
min_samples_split=3, classification=False),

    "GradientBoosting": GradientBoosting(n_estimators=100,
learning_rate=0.05, max_depth=4, min_samples_split=3,
classification=False,
checkpoint_path="/content/drive/MyDrive/checkpoint/grad2.pt",
verbose=True)
}

results_reg = {}

for name, model in models_reg.items():

    print(f"Обучаем {name}...")

    model.fit(X_reg_train, y_reg_train)

```

```

pred = model.predict(X_reg_test)

mse_val = mean_squared_error(y_reg_test, pred)
mae_val = mean_absolute_error(y_reg_test, pred)
results_reg[name] = {"MSE": mse_val, "MAE": mae_val}

print(f"{name}: MSE = {mse_val:.2f}, MAE = {mae_val:.2f}")

plt.figure(figsize=(12,5))

plt.subplot(1,2,1)

names = list(results_reg.keys())
mse_vals = [results_reg[n]["MSE"] for n in names]
sns.barplot(
    x=mse_vals,
    y=names,
    palette=["lightblue", "blue", "darkblue"]
)
plt.title("MSE")
plt.xlabel("Mean Squared Error")

plt.subplot(1,2,2)
mae_vals = [results_reg[n]["MAE"] for n in names]
sns.barplot(
    x=mae_vals,
    y=names,
    palette=["lightblue", "blue", "darkblue"]
)

```

```

)
plt.title("MAE")

plt.xlabel("Mean Absolute Error")

plt.tight_layout()

plt.show()

```

Оценка моделей на регрессии

Обучаем DecisionTree...

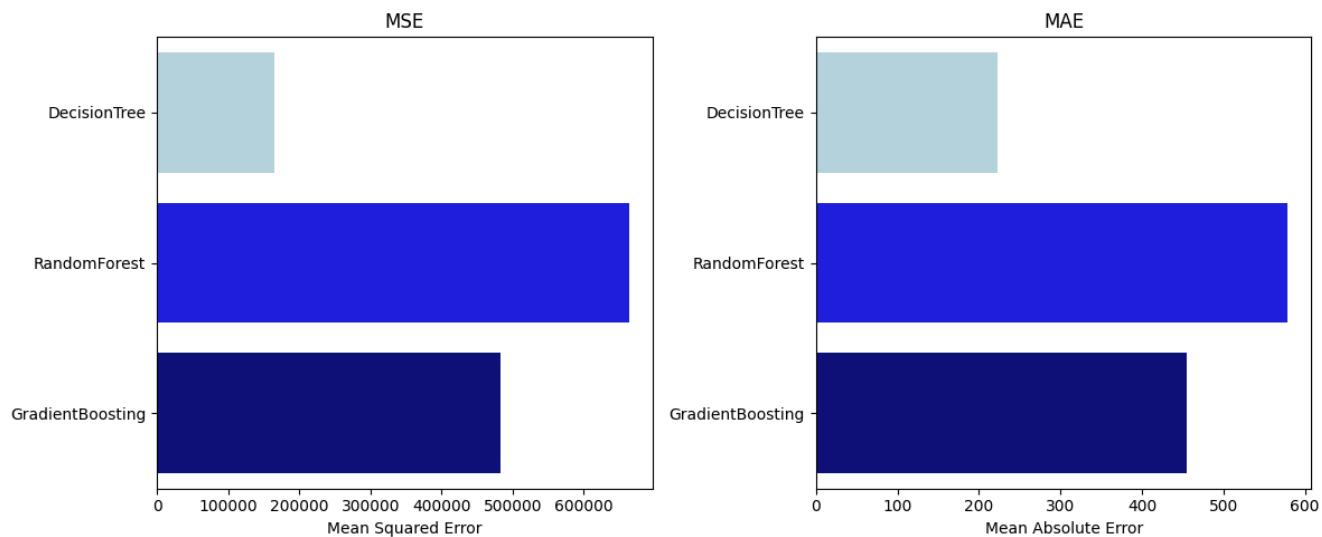
DecisionTree: MSE = 164789.49, MAE = 222.71

Обучаем RandomForest...

RandomForest: MSE = 663545.46, MAE = 579.21

Обучаем GradientBoosting...

GradientBoosting: MSE = 482762.45, MAE = 455.04



На листинге 6 представлен код обучения моделей для классификации, вывод метрик и графиков. Для повышения точности изображения CIFAR-10 размером 32x32 были преобразованы в цветовые признаки с разбиением на блоки 4x4. Для каждого блока вычисляется среднее значение пикселей по каналам R, G, B, что дает 48 признаков на изображение

Листинг 6. Код обучения моделей для классификации, счет метрик и графиков

```
X_train_raw = X_train[:10000]
y_train_raw = y_train[:10000].astype(int)
X_test_raw = X_test[:2000]
y_test_raw = y_test[:2000].astype(int)

def extract_grid_features(X, grid=4):
    X = torch.tensor(X, dtype=torch.float32).reshape(-1, 3, 32, 32)
    h = 32 // grid
    features = []

    for i in range(grid):
        for j in range(grid):
            block = X[:, :, i*h:(i+1)*h, j*h:(j+1)*h]
            features.append(block.mean(dim=(2, 3)))

    return torch.cat(features, dim=1).numpy()
```

```

X_train_flat = extract_grid_features(X_train_raw, grid=4) # 48
признаков

X_test_flat = extract_grid_features(X_test_raw, grid=4)

print("Количество признаков:", X_train_flat.shape[1])

models = {

    "DecisionTree": DecisionTree(max_depth=12, min_samples_split=10,
classification=True),

    "RandomForest": RandomForest(n_estimators=40, max_depth=10,
classification=True),

    "GradientBoosting": GradientBoosting(n_estimators=10,
learning_rate=0.15, max_depth=3, classification=True,
checkpoint_path="/content/drive/MyDrive/checkpoints/gb_class10.pt",
verbose=True)
}

results = {}

total = len(models)

for i, (name, model) in enumerate(models.items(), 1):

    print(f"Обучаем {name}...")

    model.fit(X_train_flat, y_train_raw)

    pred = model.predict(X_test_flat)

    if torch.is_tensor(pred):

        pred = pred.numpy()

    pred = np.asarray(pred).flatten()

```

```

acc = accuracy_score(y_test_raw, pred)

results[name] = acc

print(acc)

plt.figure(figsize=(10, 5))

bars = sns.barplot(
    x=list(results.values()),
    y=list(results.keys()),
    palette=["lightblue", "blue", "darkblue"]
)

plt.title("CIFAR-10: Accuracy моделей", fontsize=16, pad=20)
plt.xlabel("Accuracy")
plt.xlim(0.1, 0.5)
plt.grid(axis='x', alpha=0.3)

for i, (name, acc) in enumerate(results.items()):
    plt.text(acc + 0.005, i, f'{acc:.3f}', va='center',
fontweight='bold', fontsize=12)

plt.show()

```

Оценка моделей на классификации

Количество признаков: 48

Обучаем DecisionTree...

0.2205

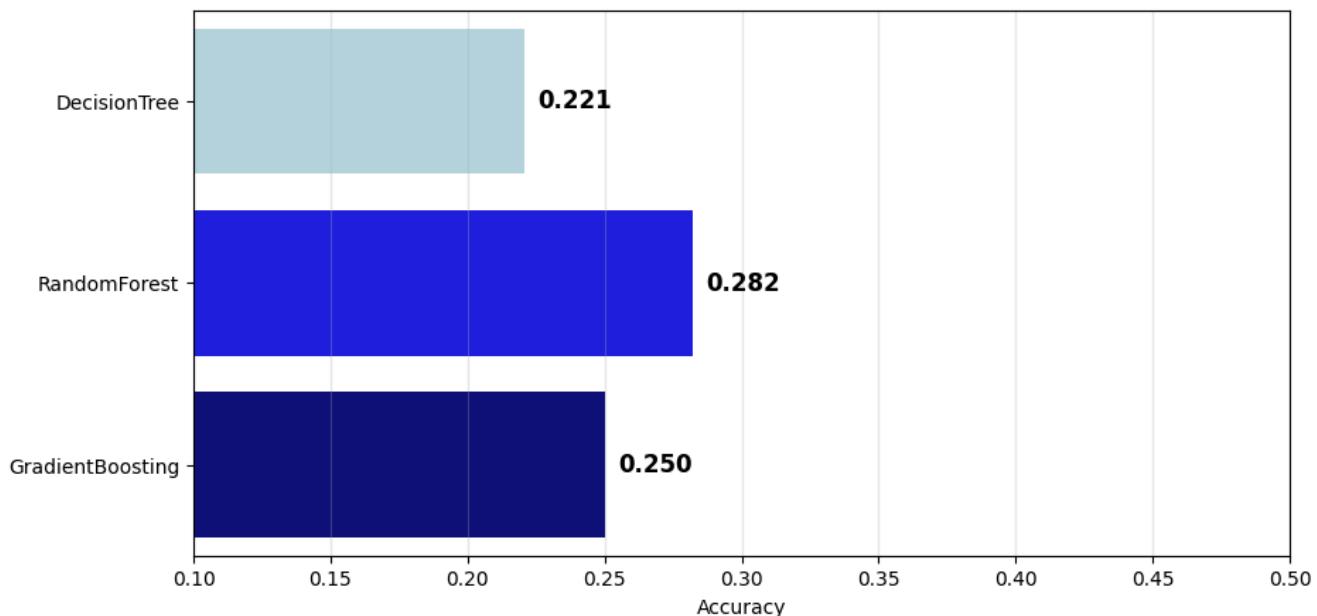
Обучаем RandomForest...

0.282

Обучаем GradientBoosting...

0.25

CIFAR-10: Accuracy моделей



Вывод

Регрессия:

Наилучший результат в регрессии мы получили от Дерева решений. Оно использует все признаки сразу, может точно подстроиться под структуру данных. Одно дерево уже дает хорошую аппроксимацию.

При этом худший результат показал Случайный лес. Так как каждое дерево обучается на случайной подвыборке данных и признаков, отдельные деревья

делают разные, часто грубые аппроксимации, а часть деревьев ошибается в разные стороны, то мы получаем переусредненее, если одно дерево уже хорошо предсказывает целевое значение. Модель слишком сглаживает зависимости

Чуть лучше показал результат Градиентный бустинг, но в данных есть шум и необъяснимые по параметрам выбросы или падения, они создают шум, под который модель начинает подгонять аппроксимацию

Классификация:

Наилучший результат в классификации показал Случайный лес. Он строит множество деревьев на разных подвыборках признаков и данных, а затем усредняет их решения. Это снижает влияние шума и случайных особенностей обучающей выборки, делая предсказания более устойчивыми и точными.

Одиночное Дерево решений показало худший результат, так как оно сильно зависит от конкретного разбиения признаков и легко переобучается на шумных данных. Предсказания становятся жёсткими и нестабильными на тестовой выборке.

Градиентный бустинг показал промежуточный результат. Он обучается последовательно, исправляя ошибки предыдущих деревьев, но признаки (средние цвета блоков) теряют часть информации о форме и текстуре объектов, а шум в данных ограничивает точность модели. В итоге улучшение происходит лишь частично, и модель не превосходит Случайный лес.