Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Методы машинного обучения

ОТЧЁТ

к лабораторной работе №4-5

на тему

Классификация данных. Деревья принятия решений

Студент Маталыга Е.А.

Преподаватель Воронина А.М.

Минск 2025

Задание:

1. Импортировать и подготовить к работы датасет бинарной классификации.
2. Создать модель бинарной классификации на основе модели логистической регрессии.
3. Создать модель бинарной классификации на основе SVM.
4. Создать модель бинарной классификации на основе CART с критериями останова. Проверить работу самописного дерева.
5. Оценить качество созданных моделей.

Выполнение:

Ссылка на полный код и датасет.

В качестве датасета были использованы данные по выдаче кредитов. Целевая переменная принимает значения 0 или 1, т.е. кредит выдан или нет.

Датасет был очищен от пропусков и выбросов, категориальные данные закодированы и весь датасет стандартизирован по тем же методам, которые были использованы в работе с разведовательным анализом данных.

Построение моделей. Модель логистической регрессии предсказывает вероятность принадлежности к классу с помощью сигмоидной функции, а само обучение основано на алгоритме градиентного спуска.

```
class LogisticRegression:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.loss_history = []
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
    def fit(self, X, y):
        X_with_bias = np.c_[np.ones((X.shape[0], 1)), X]
        y = y.reshape(-1, 1)
        n_features = X_with_bias.shape[1]
        self.weights = np.zeros((n_features, 1))

        for i in range(self.n_iterations):
```

```
                z = np.dot(X_with_bias, self.weights)
                y_pred = self.sigmoid(z)
                loss = -1/len(y) * np.sum(y * np.log(y_pred) + (1
- y) * np.log(1 - y_pred))
                self.loss_history.append(loss)
                gradient = np.dot(X_with_bias.T, (y_pred - y)) /
len(y)
                self.weights -= self.learning_rate * gradient


    model        =        LogisticRegression(learning_rate=0.1,
n_iterations=1000)
    model.fit(X_train.values, y_train.values)
```

Построение моделей. Метод опорных векторов (SVM) ищет гиперплоскость, которая максимизирует зазор между классами, т.е. минимизировать ошибку классификации.

Обучение происходит в два вложенных цикла. Внешний цикл выполняет заданное количество итераций, а внутренний цикл проходит по всем образцам обучающей выборки. Для каждого образца вычисляется margin – произведение метки класса на скалярное произведение вектора признаков и весов. Если margin ≥ 1, образец считается правильно классифицированным с достаточным запасом, если margin < 1, выполняется полное обновление весов

Код обучения выглядит следующим образом:

```
    def fit(self, X, y):
      X_with_bias = self.add_bias(X)
      n_samples, n_features = X_with_bias.shape
      self.w = np.zeros(n_features)
      for iteration in range(self.n_iters):
        total_loss = 0
        for i in range(n_samples):
          margin = y[i] * np.dot(X_with_bias[i], self.w)
          if margin >= 1:
            self.w -= self.lr * (2 * self.lambda_param * self.w)
          else:
            self.w -= self.lr * (2 * self.lambda_param * self.w
- y[i] * X_with_bias[i])
            total_loss += 1 - margin
```

```
        loss = total_loss / n_samples + self.lambda_param *
np.sum(self.w ** 2)
        self.loss_history.append(loss)
        if iteration % 100 == 0:
         print(f"Iteration {iteration}, Loss: {loss:.4f}")
```

Построение моделей. Дерево решений CART строит бинарное дерево решений. Для задачи классификации минимизируется критерий Джинни. Для задачи регрессии – MSE.

Алгоритм перебирает все возможные признаки и пороговые значения для нахождения наилучшего разделения данных. Пороговые значения вычисляются как средние между соседними уникальными значениями признака.

```
    def _best_split(self, X, y):
        features = X.columns
        min_cost_function = np.inf
        best_feature, best_threshold = None, None
        method = self._mse if self.regression else
self._gini_impurity
        for feature in features:
            unique_feature_values = np.unique(X[feature])

            for i in range(1, len(unique_feature_values)):
                current_value = unique_feature_values[i]
                previous_value = unique_feature_values[i-1]
                threshold = (current_value + previous_value) / 2
                left_indexes, right_indexes = self._split_df(X,
y, feature, threshold)
                left_labels, right_labels = y.loc[left_indexes],
y.loc[right_indexes]
                current_J = self._cost_function(left_labels,
right_labels, method)
                if current_J <= min_cost_function:
                    min_cost_function = current_J
                    best_feature = feature
                    best_threshold = threshold
        return best_feature, best_threshold
```

Дерево строится рекурсивно с помощью метода _grow_tree, который на каждом шаге проверяет условия остановки и либо создаёт листовой узел, либо продолжает разделение данных. В качестве критериев останова используются максимальная глубина дерева и минимальное количество объектов в листе.

```
    def _grow_tree(self, X, y, depth=0):
        current_num_samples = y.size
        X, y = self._set_df_type(X, y, np.float128)
        if any(self._stopping_conditions(y, depth, current_num_samples)):
            RTi = self._node_error_rate(y, method)
            leaf_node = f'{self._leaf_node(y)} | error_rate {RTi}'
            return leaf_node
```

Алгоритм включает метод cost-complexity pruning для предотвращения переобучения. Этот метод находит и удаляет слабые поддеревья, которые вносят незначительный вклад в точность предсказаний. Алгоритм рекурсивно обходит дерево и для каждого узла вычисляет его эффективность. После нахождения слабейшего узла происходит его обрезка.

```
    def cost_complexity_pruning_path(self, X: pd.DataFrame, y: pd.Series):
        tree = self._grow_tree(X, y)   # строим полное дерево
        tree_error_rate, _ = self._tree_error_rate_info(tree, [])
        error_rates = [tree_error_rate]
        ccp_alpha_list = [0.0]
        while not self._is_leaf_node(tree):
            initial_node = [None, np.inf]
            weakest_node, ccp_alpha = self._find_weakest_node(tree, initial_node)
            tree = self._prune_tree(tree, weakest_node)
            tree_error_rate, _ = self._tree_error_rate_info(tree, [])
            error_rates.append(tree_error_rate)
            ccp_alpha_list.append(ccp_alpha)
        return np.array(ccp_alpha_list), np.array(error_rates)
```

Для классификации алгоритм рекурсивно проходит по дереву, сравнивая значения признаков с пороговыми значениями в узлах, пока не достигнет листового узла.

```
def _traverse_tree(self, sample, tree):
    if self._is_leaf_node(tree):
        leaf, *_ = tree.split()
        return leaf
    decision_node = next(iter(tree))
    left_node, right_node = tree[decision_node]
    feature, other = decision_node.split(' <=')
    threshold, *_ = other.split()
    feature_value = sample[feature]
```

Итоговое построенное дерево:

```
Condition: previous_loan_defaults_on_file <= 0.5
  as_leaf 0 error_rate 0.345679012345679
  ├── Left subtree:
  │   Condition: loan_percent_income <= 0.245
  │     as_leaf 0 error_rate 0.24398463976746668
  │     ├── Left subtree:
  │     │   Condition: loan_int_rate <= 1.079222681885569
  │     │     as_leaf 0 error_rate 0.1782117332329152
  │     │     ├── Left subtree:
  │     │     │   Condition: person_income <= -1.2717965060756409
  │     │     │     as_leaf 0 error_rate 0.10746284566031188
  │     │     │     ├── Left subtree:
  │     │     │     │   Node: 1 error_rate 0.007353988603988605
  │     │     │     └── Right subtree:
  │     │     │         Node: 0 error_rate 0.08896733501237478
  │     │     └── Right subtree:
  │     │         Node: 1 error_rate 0.03560355079850493
  │     └── Right subtree:
```

```
│           Condition: home_RENT <= 0.5
│             as_leaf 1 error_rate 0.0232475284461854
│             ├── Left subtree:
│             │   Node: 0 error_rate 0.010891677122542867
│             └── Right subtree:
│               Node: 1 error_rate 0.0019199375825231127
└── Right subtree:
    Node: 0 | error_rate 0.0
```

TP=1417, TN=6724, FP=276, FN=583

Accuracy: 0.9046

Sklearn Tree Accuracy: 0.9109

Precision: 0.8370

Recall: 0.7085

F: 0.7674

Оценка качества моделей

Для оценки классификаторов используются метрики:

- Accuracy - общая точность
- Precision - точность положительного класса
- Recall - полнота положительного класса
- F-мера - гармоническое среднее precision и recall

Результаты по моделям:

- Логистическая регрессия: Accuracy = 0.8893
- SVM: Accuracy = 0.8889
- CART: Accuracy = 0.9046

Дерево решений CART показало наилучший результат.