Assignment 4

November 12, 2023

1 Late Days Used : 2 || CS 57300: Assignment 4

Name: Ishaan Roychowdhury
Date: November 2, 2023

1.1 1) Preprocessing

[1]:
```python
import pandas as pd

df = pd.read_csv('dating-full.csv', nrows=6500)
```

[2]:
```python
print("Number of rows:", df.shape[0])
```

```
Number of rows: 6500
```

[3]:
```python
#Taken from Assignment 2 (I got full points there, so this method works␣
↪correctly)
def make_map(column_data):
    unique_values_list = sorted(column_data.unique())
    value_to_integer_map = {value: index for index, value in␣
↪enumerate(unique_values_list)}
    return value_to_integer_map
def encode_column(column_data, value_to_integer_map):
    return column_data.map(value_to_integer_map)
def normalize_columns(df, columns):
    totals = df[columns].sum(axis=1)
    for col in columns:
        df[col] /= totals
    return df
```

[4]:
```python
df.drop(columns=['race', 'race_o', 'field'], inplace=True)
gender_map = make_map(df['gender'])
df['gender'] = encode_column(df['gender'], gender_map)
participant_cols = [
    'attractive_important', 'sincere_important', 'intelligence_important',
    'funny_important', 'ambition_important', 'shared_interests_important'
```

```
]
partner_cols = [
    'pref_o_attractive', 'pref_o_sincere', 'pref_o_intelligence',
    'pref_o_funny', 'pref_o_ambitious', 'pref_o_shared_interests'
]
df = normalize_columns(df, participant_cols)
df = normalize_columns(df, partner_cols)
columns_ranges = {
    'age': [18, 58],
    'age_o': [18, 58],
    'importance_same_race': [0, 10],
    'importance_same_religion': [0, 10],
    'pref_o_attractive': [0, 1],
    'pref_o_sincere': [0, 1],
    'pref_o_intelligence': [0, 1],
    'pref_o_funny': [0, 1],
    'pref_o_ambitious': [0, 1],
    'pref_o_shared_interests': [0, 1],
    'attractive_important': [0, 1],
    'sincere_important': [0, 1],
    'intelligence_important': [0, 1],
    'funny_important': [0, 1],
    'ambition_important': [0, 1],
    'shared_interests_important': [0, 1],
    'attractive': [0, 10],
    'sincere': [0, 10],
    'intelligence': [0, 10],
    'funny': [0, 10],
    'ambition': [0, 10],
    'attractive_partner': [0, 10],
    'sincere_partner': [0, 10],
    'intelligence_parter': [0, 10],
    'funny_partner': [0, 10],
    'ambition_partner': [0, 10],
    'shared_interests_partner': [0, 10],
    'sports': [0, 10],
    'tvsports': [0, 10],
    'exercise': [0, 10],
    'dining': [0, 10],
    'museums': [0, 10],
    'art': [0, 10],
    'hiking': [0, 10],
    'gaming': [0, 10],
    'clubbing': [0, 10],
    'reading': [0, 10],
    'tv': [0, 10],
    'theater': [0, 10],
    'movies': [0, 10],
    'concerts': [0, 10],
    'music': [0, 10],
    'shopping': [0, 10],
    'yoga': [0, 10],
    'interests_correlate': [-1, 1],
    'expected_happy_with_sd_people': [0, 10],
    'like': [0, 10],
}
for column_name, (minimum_value, maximum_value) in columns_ranges.items():
    clipped_series = df[column_name].clip(lower=minimum_value,␣
 ↪upper=maximum_value)
    binned_series = pd.cut(clipped_series, bins=2, labels=[0, 1],␣
 ↪include_lowest=True)
    df[column_name] = binned_series
    sorted_bin_counts = df[column_name].value_counts().sort_index()

print(df)
df.to_csv('dating-updated.csv', index=False)
test_set = df.sample(frac=0.2, random_state=47)
training_set = df.drop(test_set.index)
test_set.to_csv('testSet.csv', index=False)
training_set.to_csv('trainingSet.csv', index=False)
```

```
      gender age age_o  samerace importance_same_race  \
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
…        …   ..    …        …                   …
6495       1   0     0         1                    1
6496       1   0     0         1                    1
6497       1   0     0         0                    1
6498       1   0     0         1                    1
6499       1   0     0         1                    1

     importance_same_religion pref_o_attractive pref_o_sincere  \
0                           0                 0              0
1                           0                 1              0
2                           0                 0              0
3                           0                 0              0
4                           0                 0              0
…                        …                 …              …
6495                        1                 0              0
6496                        1                 0              0
6497                        1                 1              0
```

```
6498                        1                 0              0
6499                        1                 0              0

     pref_o_intelligence pref_o_funny  … theater movies concerts music  \
0                      0            0  …       0      1        1     1
1                      0            1  …       0      1        1     1
2                      0            1  …       0      1        1     1
3                      0            0  …       0      1        1     1
4                      1            0  …       0      1        1     1
…                    …          …   …      …    …      …   …
6495                   0            0  …       1      1        1     1
6496                   0            0  …       1      1        1     1
6497                   0            0  …       1      1        1     1
6498                   0            1  …       1      1        1     1
6499                   1            0  …       1      1        1     1

     shopping yoga interests_correlate expected_happy_with_sd_people like  \
0           1    0                   1                             0    1
1           1    0                   1                             0    1
2           1    0                   1                             0    1
3           1    0                   1                             0    1
4           1    0                   1                             0    1
…         …   …                 …                           …   …
6495        1    1                   1                             1    1
6496        1    1                   1                             1    1
6497        1    1                   1                             1    1
6498        1    1                   0                             1    1
6499        1    1                   1                             1    1

     decision
0           1
1           1
2           1
3           1
4           0
…         …
6495        0
6496        1
6497        1
6498        0
6499        0

[6500 rows x 50 columns]
```

1.2 2) Implement Decision Trees, Bagging and Random Forests (10 points)

[5]:
```
import pandas as pd
import numpy as np
from statistics import mode, multimode
from scipy.stats import mode
import matplotlib.pyplot as plt

def trees(trainingDataFilename, testDataFilename, modelIdx):
    def decisionTree(trainingSet, testSet):
        def gini_full(y):
            m = len(y)
            y_list = list(y)
            unique_elements = np.unique(y)
            sum_of_squares = 0
            for element in unique_elements:
                count = y_list.count(element)
                proportion = count / m
                squared_proportion = proportion ** 2
                sum_of_squares += squared_proportion
            gini_impurity = 1.0 - sum_of_squares
            return gini_impurity

        def split_score_current_best_split(X, y):
            split_score_current_best = float('inf')
            split_score_current_best_split = None
            bestY, bestX = None, None
            counter = 0
            columns = X.columns.tolist()
            while counter < len(columns):
                column = columns[counter]
                all_possible_t = X[column].unique()
                t_val_index = 0
                while t_val_index < len(all_possible_t):
                    t_val = all_possible_t[t_val_index]
                    left_y = y[X[column] < t_val]
                    right_y = y[X[column] >= t_val]
                    left_gini = gini_full(left_y)
                    right_gini = gini_full(right_y)
                    score = (len(left_y) / len(y)) * left_gini + (len(right_y) /
 ↪ len(y)) * right_gini
                    if score < split_score_current_best:
                        split_score_current_best = score
                        split_score_current_best_split = (column, t_val)
                        bestY, bestX = left_y, right_y
                    t_val_index += 1
                counter += 1
            return split_score_current_best, split_score_current_best_split,␣
↪bestY, bestX

        def build_tree(X, y, depth=0):
            conditions = {
                1: lambda: depth == 8,
                2: lambda: len(y) < 50,
                3: lambda: len(np.unique(y)) == 1
            }
            for key, condition in conditions.items():
                if condition():
                    return {'node': {'prediction': y.mode()[0]}}

            _, (column, t_val), left_y, right_y =␣
↪split_score_current_best_split(X, y)
            if not column:
                return {'node': {'prediction': y.mode()[0]}}

            left_tree = build_tree(X[X[column] < t_val], left_y, depth + 1)
            right_tree = build_tree(X[X[column] >= t_val], right_y, depth + 1)

            return {
                'decision': {
                    'column': column,
                    'threshold': t_val
                },
                'branches': {
                    'left': left_tree,
                    'right': right_tree
                }
            }

        def predict(tree, x):
            while True:
                if 'node' in tree:
                    return tree['node']['prediction']
                column, t_val = tree['decision']['column'],␣
↪tree['decision']['threshold']
                tree = tree['branches']['left'] if x[column] < t_val else␣
↪tree['branches']['right']

        train_data = pd.read_csv(trainingSet)
        test_data = pd.read_csv(testSet)
        X_train, y_train = train_data.drop('decision', axis=1),␣
↪train_data['decision']
        X_test, y_test = test_data.drop('decision', axis=1),␣
↪test_data['decision']
```

```
        tree = build_tree(X_train, y_train)
        train_predictions = X_train.apply(lambda x: predict(tree, x), axis=1)
        train_accuracy = (train_predictions == y_train).mean()
        test_predictions = X_test.apply(lambda x: predict(tree, x), axis=1)
        test_accuracy = (test_predictions == y_test).mean()
        print(f'Training Accuracy DT: {train_accuracy}')
        print(f'Test Accuracy DT: {test_accuracy}')
        return train_accuracy, test_accuracy

    import pandas as pd
    import numpy as np
    from scipy.stats import mode as scipy_mode

    def bagging(trainingSet, testSet):
        num_trees=30
        def calculate_gini_impurity(y):
            total_items = len(y)
            y_list = list(y)
            unique_elements = np.unique(y)
            impurity_sum = 0
            for element in unique_elements:
                element_count = y_list.count(element)
                element_proportion = element_count / total_items
                impurity_sum += element_proportion ** 2
            return 1.0 - impurity_sum

        def find_optimal_split(X, y):
            best_score_bag = float('inf')
            best_split_bag = None
            bestYbag, bestXbag = None, None

            col_index = 0
            while col_index < len(X.columns):
                column = X.columns[col_index]
                total_t = X[column].unique()
                t_count = 0
                while t_count < len(total_t):
                    current_t = total_t[t_count]
                    left_y, right_y = y[X[column] < current_t], y[X[column] >=␣
↪current_t]
                    gini_left, gini_right = calculate_gini_impurity(left_y),␣
↪calculate_gini_impurity(right_y)
                    combined_score = (len(left_y) / len(y)) * gini_left +␣
↪(len(right_y) / len(y)) * gini_right
                    if combined_score < best_score_bag:
                        best_score_bag = combined_score
                        best_split_bag = (column, current_t)
                        bestYbag, bestXbag = left_y, right_y
                    t_count += 1
                col_index += 1
            return best_score_bag, best_split_bag, bestYbag, bestXbag

        def create_decision_tree(X, y, depth=0):
            if depth == 8 or len(y) < 50 or len(np.unique(y)) == 1:
                return {'node': {'prediction': scipy_mode(y).mode[0]}}

            _, (split_column, split_value), left_y, right_y =␣
↪find_optimal_split(X, y)
            if not split_column:
                return {'node': {'prediction': scipy_mode(y).mode[0]}}

            left_branch = create_decision_tree(X[X[split_column] <␣
↪split_value], left_y, depth + 1)
            right_branch = create_decision_tree(X[X[split_column] >=␣
↪split_value], right_y, depth + 1)

            return {
                'decision_criteria': {
                    'column': split_column,
                    'current_t': split_value
                },
                'subtrees': {
                    'left': left_branch,
                    'right': right_branch
                }
            }

        def tree_prediction(tree, x):
            while 'node' not in tree:
                when_to_split = tree['decision_criteria']
                tree = tree['subtrees']['left'] if x[when_to_split['column']] <␣
↪when_to_split['current_t'] else tree['subtrees']['right']
            return tree['node']['prediction']

        def generate_bootstrap_sample(X, y):
            sample_indices = np.random.randint(0, len(X), len(X))
            return X.iloc[sample_indices], y.iloc[sample_indices]

        def ensemble_prediction(trees, x):
            predictions = [tree_prediction(tree, x) for tree in trees]
            return scipy_mode(predictions).mode[0]

        train_dataset = pd.read_csv(trainingSet)
        test_dataset = pd.read_csv(testSet)
```

```
        X_train, y_train = train_dataset.drop('decision', axis=1),␣
↪train_dataset['decision']
        X_test, y_test = test_dataset.drop('decision', axis=1),␣
↪test_dataset['decision']
        decision_trees = []
        tree_index = 0
        while tree_index < num_trees:
            X_sample, y_sample = generate_bootstrap_sample(X_train, y_train)
            tree = create_decision_tree(X_sample, y_sample)
            decision_trees.append(tree)
            tree_index += 1
        train_predictions = X_train.apply(lambda x:␣
↪ensemble_prediction(decision_trees, x), axis=1)
        test_predictions = X_test.apply(lambda x:␣
↪ensemble_prediction(decision_trees, x), axis=1)
        accuracy_train = np.mean(train_predictions == y_train)
        accuracy_test = np.mean(test_predictions == y_test)
        print(f'Train Accuracy BT: {accuracy_train}')
        print(f'Test Accuracy BT: {accuracy_test}')
        return accuracy_train, accuracy_test

    import pandas as pd
    import numpy as np
    from statistics import mode

    def randomForests(trainingSet, testSet):
        num_trees = 30
        max_tree_depth = 8
        min_samples = 50

        def calculate_gini_impurity(y):
            total_count = len(y)
            y_list = list(y)
            impurity_sum = 0
            for element in np.unique(y):
                element_count = y_list.count(element)
                element_proportion = element_count / total_count
                impurity_sum += element_proportion ** 2
            return 1.0 - impurity_sum

        def optimal_split(X, y):
            best_score_forest = float('inf')
            best_score_forest_deet = None
            forestY, forestX = None, None

            featForest = np.random.choice(X.columns, int(np.sqrt(len(X.
↪columns))), replace=False)
```

```
        for feature in featForest:
            for threshold in X[feature].unique():
                left, right = y[X[feature] < threshold], y[X[feature] >=␣
↪threshold]
                curr_score = (len(left) / len(y)) *␣
↪calculate_gini_impurity(left) + (len(right) / len(y)) *␣
↪calculate_gini_impurity(right)
                if curr_score < best_score_forest:
                    best_score_forest = curr_score
                    best_score_forest_deet = (feature, threshold)
                    forestY, forestX = left, right

        return best_score_forest, best_score_forest_deet, forestY, forestX

    def generate_tree(X, y, depth=0, max_depth=max_tree_depth,␣
↪min_split=min_samples):
        if len(y) == 0:
            return {'node': {'prediction': None}}

        if len(np.unique(y)) == 1 or depth == max_depth or len(y) <␣
↪min_split:
            return {'node': {'prediction': mode(y)}}

        _, (split_feature, split_value), left_y, right_y = optimal_split(X,␣
↪y)
        if not split_feature:
            return {'node': {'prediction': mode(y)}}

        left_branch = generate_tree(X[X[split_feature] < split_value],␣
↪left_y, depth + 1)
        right_branch = generate_tree(X[X[split_feature] >= split_value],␣
↪right_y, depth + 1)

        return {
            'when_to_split': {
                'feature': split_feature,
                'value': split_value
            },
            'branches': {
                'left': left_branch,
                'right': right_branch
            }
        }


    def predict_from_tree(tree, instance):
            while 'node' not in tree:
                criteria = tree['when_to_split']
                tree = tree['branches']['left'] if␣
↪instance[criteria['feature']] < criteria['value'] else␣
↪tree['branches']['right']
            return tree['node']['prediction']

        #WITH REPLACEMENT, hence setting to TRUE
        def create_forest(X, y, trees_count):
            forest = []
            for _ in range(trees_count):
                sample_Forest = np.random.choice(X.index, size=len(X),␣
↪replace=True)
                X_sample, y_sample = X.loc[sample_Forest], y.loc[sample_Forest]
                tree = generate_tree(X_sample, y_sample)
                forest.append(tree)
            return forest

        def predict_with_forest(forest, X):
            predictions = [[predict_from_tree(tree, x) for tree in forest] for␣
↪_, x in X.iterrows()]
            mode_predictions = [mode(pred) for pred in predictions]
            return mode_predictions

        train_dataset = pd.read_csv(trainingSet)
        test_dataset = pd.read_csv(testSet)
        X_train, y_train = train_dataset.drop('decision', axis=1),␣
↪train_dataset['decision']
        X_test, y_test = test_dataset.drop('decision', axis=1),␣
↪test_dataset['decision']
        forest = create_forest(X_train, y_train, num_trees)
        train_accuracy = np.mean(predict_with_forest(forest, X_train) ==␣
↪y_train)
        test_accuracy = np.mean(predict_with_forest(forest, X_test) == y_test)
        print(f'Training Accuracy RF: {train_accuracy}')
        print(f'Testing Accuracy RF: {test_accuracy}')

        return train_accuracy, test_accuracy

    if modelIdx == 1:
        return decisionTree(trainingDataFilename, testDataFilename)
    elif modelIdx == 2:
        return bagging(trainingDataFilename, testDataFilename)
    elif modelIdx == 3:
        return randomForests(trainingDataFilename, testDataFilename)
    else:
        raise ValueError("Please man put the right number.")
```

[6]:
```
trees('trainingSet.csv', 'testSet.csv', 1)
```

```
Training Accuracy DT: 0.7788461538461539
Test Accuracy DT: 0.7092307692307692
```

[6]:
```
(0.7788461538461539, 0.7092307692307692)
```

[7]:
```
trees('trainingSet.csv', 'testSet.csv', 2)
```

```
Train Accuracy BT: 0.7965384615384615
Test Accuracy BT: 0.75
```

[7]:
```
(0.7965384615384615, 0.75)
```

[8]:
```
trees('trainingSet.csv', 'testSet.csv', 3)
```

```
Training Accuracy RF: 0.7751923076923077
Testing Accuracy RF: 0.7323076923076923
```

[8]:
```
(0.7751923076923077, 0.7323076923076923)
```

1.3 3) a) The Influence of Tree Depth on Classifier Performance (10 points)

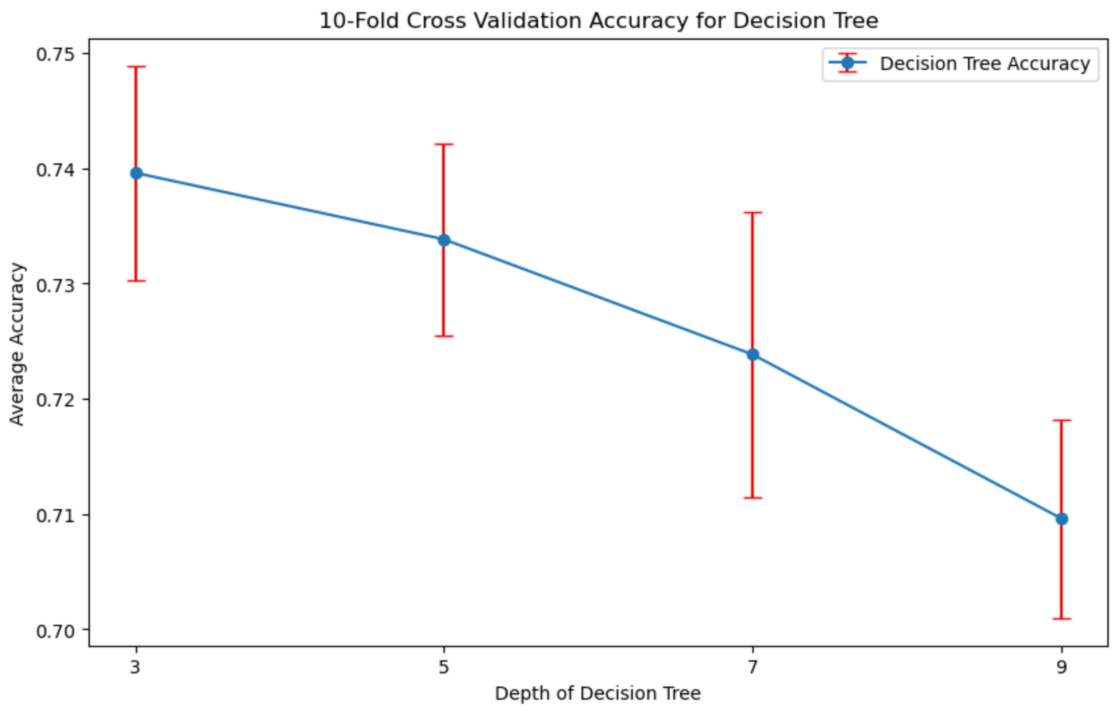
[18]:
```
import pandas as pd
import numpy as np
from statistics import mode, multimode
from scipy.stats import mode
def treesForQ3_with_Depth(trainingDataFilename, testDataFilename, modelIdx,␣
 ↪max_depth =8):
    def decisionTreeQ3(trainingSet, testSet, max_depth):
        def gini_full(y):
            m = len(y)
            y_list = list(y)
            unique_elements = np.unique(y)
            sum_of_squares = 0
            for element in unique_elements:
                count = y_list.count(element)
                proportion = count / m
                squared_proportion = proportion ** 2
                sum_of_squares += squared_proportion
            return 1.0 - sum_of_squares

        def split_score_current_best_split(X, y):
            split_score_current_best = float('inf')
            split_score_current_best_split = None
            best_left_y, best_right_y = None, None
            counter = 0
            columns = X.columns.tolist()
            while counter < len(columns):
                column = columns[counter]
                all_possible_t = X[column].unique()
                t_val_index = 0
                while t_val_index < len(all_possible_t):
                    t_val = all_possible_t[t_val_index]
                    left_y = y[X[column] < t_val]
                    right_y = y[X[column] >= t_val]
                    left_gini = gini_full(left_y)
                    right_gini = gini_full(right_y)
                    score = (len(left_y) / len(y)) * left_gini + (len(right_y) /
↪ len(y)) * right_gini
                    if score < split_score_current_best:
                        split_score_current_best = score
                        split_score_current_best_split = (column, t_val)
                        best_left_y, best_right_y = left_y, right_y
                    t_val_index += 1
                counter += 1
            return split_score_current_best, split_score_current_best_split,␣
↪best_left_y, best_right_y

        def build_tree(X, y, depth=0, max_depth=max_depth):
            index = 0
            conditions = [
                lambda: depth == max_depth,
                lambda: len(y) < 50,
                lambda: len(np.unique(y)) == 1
            ]

            while index < len(conditions):
                if conditions[index]():
                    return {'node': {'prediction': y.mode()[0]}}
                index += 1

            _, (column, t_val), left_y, right_y =␣
↪split_score_current_best_split(X, y)
            if not column:
                return {'node': {'prediction': y.mode()[0]}}

            left_tree = build_tree(X[X[column] < t_val], left_y, depth + 1)
            right_tree = build_tree(X[X[column] >= t_val], right_y, depth + 1)

            return {
                'decision': {
                    'column': column,
                    'threshold': t_val
                },
                'branches': {
                    'left': left_tree,
                    'right': right_tree
                }
            }

        def predict(tree, x):
            while True:
                if 'node' in tree:
                    return tree['node']['prediction']
                column, t_val = tree['decision']['column'],␣
↪tree['decision']['threshold']
                tree = tree['branches']['left'] if x[column] < t_val else␣
↪tree['branches']['right']

        X_train, y_train = trainingSet.drop('decision', axis=1),␣
↪trainingSet['decision']
        X_test, y_test = testSet.drop('decision', axis=1), testSet['decision']
        tree = build_tree(X_train, y_train)
        train_predictions = X_train.apply(lambda x: predict(tree, x), axis=1)
        train_accuracy = (train_predictions == y_train).mean()
        test_predictions = X_test.apply(lambda x: predict(tree, x), axis=1)
        test_accuracy = (test_predictions == y_test).mean()
        return train_accuracy, test_accuracy

    import pandas as pd
    import numpy as np
    from statistics import multimode
    from scipy.stats import mode as scipy_mode

    def baggingQ3(trainingSet, testSet, max_depth):
        num_trees = 30

        def calculate_gini_impurity(y):
            total_items = len(y)
            y_list = list(y)
            unique_elements = np.unique(y)
            impurity_sum = 0
            for element in unique_elements:
                element_count = y_list.count(element)
                element_proportion = element_count / total_items
                impurity_sum += element_proportion ** 2
            return 1.0 - impurity_sum
        def find_optimal_split(X, y):
            best_score_bag = float('inf')
            best_split_bag = None
            bestYbag, bestXbag = None, None

            col_index = 0
            while col_index < len(X.columns):
                column = X.columns[col_index]
                total_t = X[column].unique()
                t_count = 0
                while t_count < len(total_t):
                    current_t = total_t[t_count]
                    left_y, right_y = y[X[column] < current_t], y[X[column] >=␣
    ↪current_t]
                    gini_left, gini_right = calculate_gini_impurity(left_y),␣
    ↪calculate_gini_impurity(right_y)
                    combined_score = (len(left_y) / len(y)) * gini_left +␣
    ↪(len(right_y) / len(y)) * gini_right
                    if combined_score < best_score_bag:
                        best_score_bag = combined_score
                        best_split_bag = (column, current_t)
                        bestYbag, bestXbag = left_y, right_y
                    t_count += 1
                col_index += 1
            return best_score_bag, best_split_bag, bestYbag, bestXbag

        def create_decision_tree(X, y, depth=0):
            if depth == max_depth or len(y) < 50 or len(np.unique(y)) == 1:
                return {'node': {'prediction': scipy_mode(y).mode[0]}}

            _, (split_column, split_value), left_y, right_y =␣
    ↪find_optimal_split(X, y)
            if not split_column:
                return {'node': {'prediction': scipy_mode(y).mode[0]}}

            left_branch = create_decision_tree(X[X[split_column] <␣
    ↪split_value], left_y, depth + 1)
            right_branch = create_decision_tree(X[X[split_column] >=␣
    ↪split_value], right_y, depth + 1)

            return {
                'decision_criteria': {
                    'column': split_column,
                    'current_t': split_value
                },
                'subtrees': {
                    'left': left_branch,
                    'right': right_branch
                }
            }

        def tree_prediction(tree, x):
            while 'node' not in tree:
                when_to_split = tree['decision_criteria']
                tree = tree['subtrees']['left'] if x[when_to_split['column']] <␣
↪when_to_split['current_t'] else tree['subtrees']['right']
            return tree['node']['prediction']

        def generate_bootstrap_sample(X, y):
            sample_indices = np.random.randint(0, len(X), len(X))
            return X.iloc[sample_indices], y.iloc[sample_indices]

        def ensemble_prediction(trees, x):
            predictions = [tree_prediction(tree, x) for tree in trees]
            return scipy_mode(predictions).mode[0]

        X_train, y_train = trainingSet.drop('decision', axis=1),␣
↪trainingSet['decision']
        X_test, y_test = testSet.drop('decision', axis=1), testSet['decision']
        decision_trees = []
        tree_index = 0
        while tree_index < num_trees:
            X_sample, y_sample = generate_bootstrap_sample(X_train, y_train)
            tree = create_decision_tree(X_sample, y_sample, 0)
            decision_trees.append(tree)
            tree_index += 1
        train_predictions = X_train.apply(lambda x:␣
↪ensemble_prediction(decision_trees, x), axis=1)
        test_predictions = X_test.apply(lambda x:␣
↪ensemble_prediction(decision_trees, x), axis=1)
        accuracy_train = np.mean(train_predictions == y_train)
        accuracy_test = np.mean(test_predictions == y_test)
        return accuracy_train, accuracy_test

    import pandas as pd
    import numpy as np
    from statistics import mode

    def randomForestsQ3(trainingSet, testSet, max_depth):
        num_trees = 30
        max_tree_depth = 8
        min_samples = 50
        def calculate_gini_impurity(y):
            total_count = len(y)
            y_list = list(y)
            impurity_sum = 0
            for element in np.unique(y):
                element_count = y_list.count(element)
                element_proportion = element_count / total_count
                impurity_sum += element_proportion ** 2
            return 1.0 - impurity_sum

        def optimal_split(X, y):
            best_score_forest = float('inf')
            best_score_forest_deet = None
            forestY, forestX = None, None

            featForest = np.random.choice(X.columns, int(np.sqrt(len(X.
↪columns))), replace=False)
            for feature in featForest:
                for threshold in X[feature].unique():
                    left, right = y[X[feature] < threshold], y[X[feature] >=␣
↪threshold]
                    curr_score = (len(left) / len(y)) *␣
↪calculate_gini_impurity(left) + (len(right) / len(y)) *␣
↪calculate_gini_impurity(right)
                    if curr_score < best_score_forest:
                        best_score_forest = curr_score
                        best_score_forest_deet = (feature, threshold)
                        forestY, forestX = left, right
            return best_score_forest, best_score_forest_deet, forestY, forestX

        def generate_tree(X, y, depth=0):
            if len(y) == 0:
                return {'node': {'prediction': None}}
            if len(np.unique(y)) == 1 or depth == max_tree_depth or len(y) <␣
↪min_samples:
                return {'node': {'prediction': mode(y)}}
            _, (split_feature, split_value), left_y, right_y = optimal_split(X,␣
↪y)
            if not split_feature:
                return {'node': {'prediction': mode(y)}}
            left_branch = generate_tree(X[X[split_feature] < split_value],␣
↪left_y, depth + 1)
            right_branch = generate_tree(X[X[split_feature] >= split_value],␣
↪right_y, depth + 1)
            return {
                'when_to_split': {
                    'feature': split_feature,
                    'value': split_value
                },
                'branches': {
                    'left': left_branch,
                    'right': right_branch
                }
            }

        def predict_from_tree(tree, instance):
            while 'node' not in tree:
                criteria = tree['when_to_split']
                tree = tree['branches']['left'] if␣
↪instance[criteria['feature']] < criteria['value'] else␣
↪tree['branches']['right']
            return tree['node']['prediction']

        def create_forest(X, y):
            forest = []
            for _ in range(num_trees):
                sample_Forest = np.random.choice(X.index, size=len(X),␣
↪replace=True)
                X_sample, y_sample = X.loc[sample_Forest], y.loc[sample_Forest]
                tree = generate_tree(X_sample, y_sample)
                forest.append(tree)
            return forest

        def predict_with_forest(forest, X):
            predictions = [[predict_from_tree(tree, x) for tree in forest] for␣
↪_, x in X.iterrows()]
            mode_predictions = [mode(pred) for pred in predictions]
            return mode_predictions

        X_train, y_train = trainingSet.drop('decision', axis=1),␣
↪trainingSet['decision']
        X_test, y_test = testSet.drop('decision', axis=1), testSet['decision']
        forest = create_forest(X_train, y_train)
        train_accuracy = np.mean(predict_with_forest(forest, X_train) ==␣
↪y_train)
        test_accuracy = np.mean(predict_with_forest(forest, X_test) == y_test)
        return train_accuracy, test_accuracy

    if modelIdx == 1:
        return decisionTreeQ3(trainingDataFilename, testDataFilename,max_depth)
    elif modelIdx == 2:
        return baggingQ3(trainingDataFilename, testDataFilename, max_depth)
    elif modelIdx == 3:
        return randomForestsQ3(trainingDataFilename, testDataFilename,␣
↪max_depth)
    else:
        raise ValueError("Put correct idx number man.")
```

[12]:
```
training_data_path = 'trainingSet.csv'
data = pd.read_csv(training_data_path)
shuffled_data = data.sample(frac=1, random_state=18)
sampled_data = shuffled_data.sample(frac=0.5, random_state=32)

import pandas as pd
import numpy as np

def cross_validation_decision_tree(data, depths, fold_count=10):
    fold_size = len(data) // fold_count
    results = {depth: [] for depth in depths}

    for i in range(fold_count):
        test_data = data.iloc[i*fold_size:(i+1)*fold_size]
        train_data = pd.concat([data.iloc[:i*fold_size], data.
↪iloc[(i+1)*fold_size:]])

        for depth in depths:
            _, test_accuracy = treesForQ3_with_Depth(train_data, test_data, 1,␣
↪depth)
            results[depth].append(test_accuracy)

    avg_accuracies_dt = {depth: np.mean(results[depth]) for depth in depths}
    std_errors_dt = {depth: np.std(results[depth], ddof=1) / np.
↪sqrt(fold_count) for depth in depths}
    return avg_accuracies_dt, std_errors_dt, results

depths = [3, 5, 7, 9]
avg_accuracies_dt, std_errors_dt, accuracies_dt =␣
↪cross_validation_decision_tree(sampled_data, depths)
print("Average Accuracies:", avg_accuracies_dt)
print("Standard Errors:", std_errors_dt)
```

```
Average Accuracies: {3: 0.7396153846153847, 5: 0.7338461538461538, 7:
0.7238461538461538, 9: 0.7096153846153845}
Standard Errors: {3: 0.00929907066963517, 5: 0.0083441744078323, 7:
0.012334372790569943, 9: 0.008624117978893396}
```

[13]:
```
plt.figure(figsize=(10, 6))
plt.errorbar(depths, [avg_accuracies_dt[d] for d in depths],␣
↪yerr=[std_errors_dt[d] for d in depths],
             fmt='-o', ecolor='red', capsize=5, label='Decision Tree Accuracy')
```

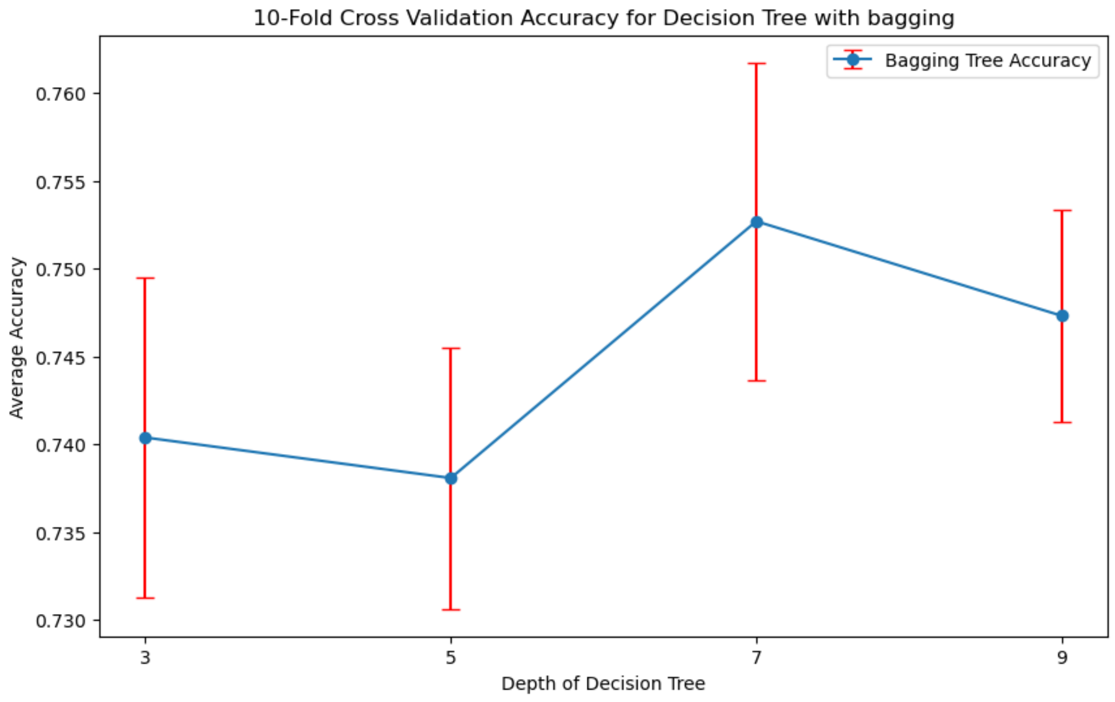
```
plt.xlabel('Depth of Decision Tree')
plt.ylabel('Average Accuracy')
plt.title('10-Fold Cross Validation Accuracy for Decision Tree')
plt.xticks(depths)
plt.legend()
plt.show()
```



[14]:
```
import pandas as pd
import numpy as np

training_data_path = 'trainingSet.csv'
data = pd.read_csv(training_data_path)
shuffled_data = data.sample(frac=1, random_state=18)
sampled_data = shuffled_data.sample(frac=0.5, random_state=32)
def cross_validation_bagging(data, depths, fold_count=10):
    fold_size = len(data) // fold_count
    results = {depth: [] for depth in depths}
    for i in range(fold_count):
        test_data = data.iloc[i * fold_size:(i + 1) * fold_size]
        train_data = data.drop(test_data.index)
        for depth in depths:
            _, test_accuracy = treesForQ3_with_Depth(train_data, test_data, 2,␣
 ↪depth)
```

```
            results[depth].append(test_accuracy)
    avg_accuracies_bag = {depth: np.mean(results[depth]) for depth in depths}
    std_errors_bag = {depth: np.std(results[depth]) / np.sqrt(fold_count) for␣
 ↪depth in depths}
    return avg_accuracies_bag, std_errors_bag, results
depths = [3, 5, 7, 9]
avg_accuracies_bag , std_errors_bag, accuracies_bag =␣
 ↪cross_validation_bagging(sampled_data, depths)
```

[15]:
```
plt.figure(figsize=(10, 6))
plt.errorbar(depths, [avg_accuracies_bag[d] for d in depths],␣
 ↪yerr=[std_errors_bag[d] for d in depths],
             fmt='-o', ecolor='red', capsize=5, label='Bagging Tree Accuracy')
plt.xlabel('Depth of Decision Tree')
plt.ylabel('Average Accuracy')
plt.title('10-Fold Cross Validation Accuracy for Decision Tree with bagging')
plt.xticks(depths)
plt.legend()
plt.show()
```

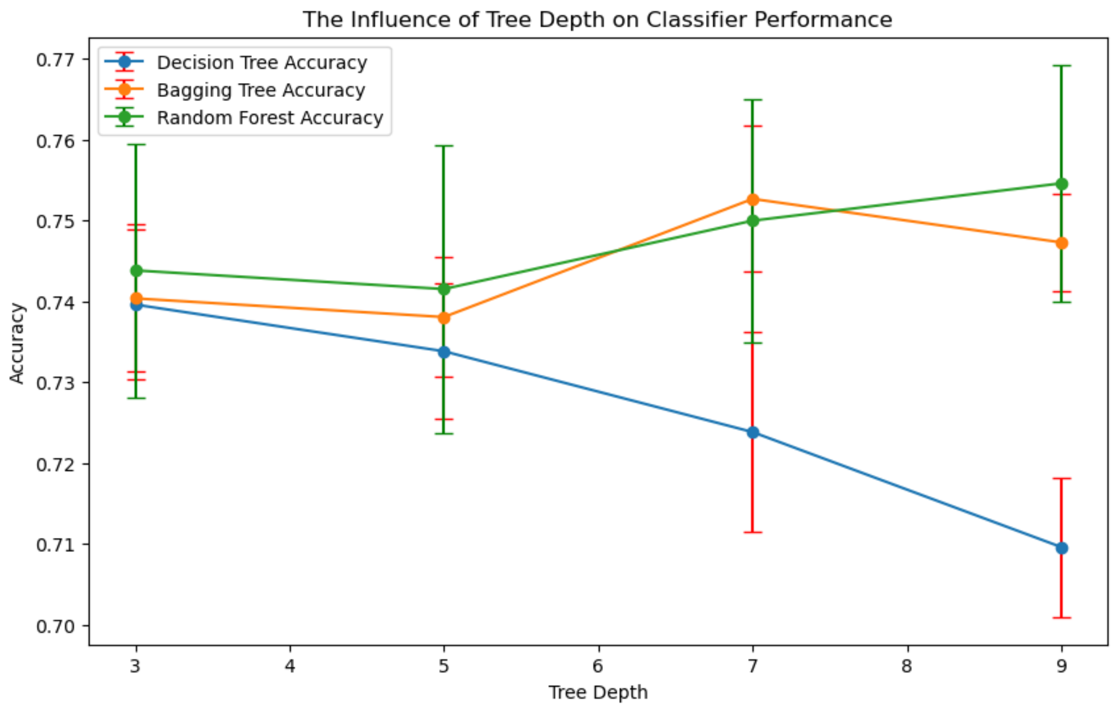


[21]:
```
import pandas as pd
import numpy as np
```

```
def cross_validation_random_forests(data, depths, fold_count=10, n_trees=30,␣
↪min_samples_split=2):
    fold_size = len(data) // fold_count
    results = {depth: [] for depth in depths}
    for i in range(fold_count):
        test_data = data.iloc[i*fold_size:(i+1)*fold_size]
        train_data = pd.concat([data.iloc[:i*fold_size], data.
↪iloc[(i+1)*fold_size:]])
        for depth in depths:
            _, test_accuracy = treesForQ3_with_Depth(train_data, test_data, 3,␣
↪depth)
            results[depth].append(test_accuracy)
    avg_accuracies_rf = {depth: np.mean(results[depth]) for depth in depths}
    std_errors_rf = {depth: np.std(results[depth]) / np.sqrt(fold_count) for␣
↪depth in depths}
    return avg_accuracies_rf, std_errors_rf, results

data = pd.read_csv('trainingSet.csv')
sampled_data = data.sample(frac=0.5, random_state=32)

depths = [3, 5, 7, 9]
avg_accuracies_random_forests, std_errors_rf, accuracies_rf =␣
↪cross_validation_random_forests(sampled_data, depths)
print("Done Sucessfully Part 3")
```

```
Done Sucessfully Part 3
```

[25]:
```
plt.figure(figsize=(10, 6))
plt.errorbar(depths, [avg_accuracies_dt[d] for d in depths],␣
↪yerr=[std_errors_dt[d] for d in depths],
             fmt='-o', ecolor='blue', capsize=5, label='Decision Tree Accuracy')
plt.errorbar(depths, [avg_accuracies_bag[d] for d in depths],␣
↪yerr=[std_errors_bag[d] for d in depths],
             fmt='-o', ecolor='yellow', capsize=5, label='Bagging Tree␣
↪Accuracy')
plt.errorbar(depths, [avg_accuracies_random_forests[d] for d in depths],␣
↪yerr=[std_errors_rf[d] for d in depths],
             fmt='-o', ecolor='green', capsize=5, label='Random Forest␣
↪Accuracy')
plt.xlabel('Tree Depth')
plt.ylabel('Accuracy')
plt.title('The Influence of Tree Depth on Classifier Performance')
plt.legend()
plt.show()
```



1.4 3) b) Hypothesis

Null Hypothesis (H0): There is no significant difference in accuracy between the Decision Tree and Bagged Trees classifiers. Alternative Hypothesis (H1): There is a significant difference in accuracy between the Decision Tree and Bagged Trees classifiers.

[26]:
```
from scipy.stats import ttest_rel
p_values = {}
for depth in depths:
    dt_accuracies = accuracies_dt[depth]
    bagging_accuracies = accuracies_bag[depth]
    t_statistic, p_value = ttest_rel(dt_accuracies, bagging_accuracies)
    p_values[depth] = p_value
```

[28]:
```
alpha = 0.05
hypothesis_results = {depth: 'Reject Null Hypothesis' if p < alpha else 'Accept␣
 ↪Null Hypothesis' for depth, p in p_values.items()}
hypothesis_results
```

[28]:
```
{3: 'Accept Null Hypothesis',
 5: 'Reject Null Hypothesis',
 7: 'Reject Null Hypothesis',
 9: 'Reject Null Hypothesis'}
```

We have chosen an alpha value of 0.05. And we reject the null hypothesis if the p-value is less than alpha. This means for depth 3, we accept the null hypothesis since p-value is more than 0.05 for this case. In all the other depths, we reject the null hypothesis since the p-value is less than 0.05. We can also see this from the graph. *To incorporate specific training set size we have run the cross validation methods and used the accuracy for each depth

---

1.5 4) a) Compare Performance of Different Models (10 points)

[29]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

file_path = 'trainingSet.csv'
data = pd.read_csv(file_path)

data_shuffled = data.sample(frac=1, random_state=18)

t_frac_val = [0.05, 0.075, 0.1, 0.15, 0.2]
dict_for_models_acc = {
    'Decision Tree': {frac: [] for frac in t_frac_val},
    'Bagged Trees': {frac: [] for frac in t_frac_val},
    'Random Forest': {frac: [] for frac in t_frac_val}
}
k = 10
fold_size = int(len(data_shuffled) / k)

for i in range(k):
    start = i * fold_size
    end = start + fold_size if i != k - 1 else len(data_shuffled)
    manual_k_fold_test = data_shuffled.iloc[start:end]
    train_data = pd.concat([data_shuffled[:start], data_shuffled[end:]])

    for frac in t_frac_val:
        manual_k_fold_train = train_data.sample(frac=frac, random_state=32)
        manual_k_fold_train.to_csv('temp_train.csv', index=False)
        manual_k_fold_test.to_csv('temp_test.csv', index=False)
        _, dt_accuracy = trees('temp_train.csv', 'temp_test.csv', 1)
        _, bagging_accuracy = trees('temp_train.csv', 'temp_test.csv', 2)
        _, rf_accuracy = trees('temp_train.csv', 'temp_test.csv', 3)
        dict_for_models_acc['Decision Tree'][frac].append(dt_accuracy)
        dict_for_models_acc['Bagged Trees'][frac].append(bagging_accuracy)
        dict_for_models_acc['Random Forest'][frac].append(rf_accuracy)

final_acc_for_all = {}
for model in dict_for_models_acc:
    final_acc_for_all[model] = {
        'avg_accuracy': [np.mean(dict_for_models_acc[model][frac]) for frac in␣
 ↪t_frac_val],
        'std_error': [np.std(dict_for_models_acc[model][frac]) / np.
 ↪sqrt(len(dict_for_models_acc[model][frac])) for frac in t_frac_val]
    }

for model in dict_for_models_acc:
    print(f"{model}:")
    for frac in t_frac_val:
        print(f"  Training fraction {frac}: {dict_for_models_acc[model][frac]}")

# Plotting the learning curves
plt.figure(figsize=(12, 8))
for model in final_acc_for_all:
    plt.errorbar(t_frac_val, final_acc_for_all[model]['avg_accuracy'],␣
 ↪yerr=final_acc_for_all[model]['std_error'],
                 fmt='-o', capsize=5, label=model)
plt.xlabel('Training Fraction (t_frac)')
plt.ylabel('Average Accuracy')
plt.title('Learning Curves for Decision Tree, Bagged Trees, and Random Forest')
plt.xticks(t_frac_val)
plt.legend()
plt.show()
```

```
Training Accuracy DT: 0.8076923076923077
Test Accuracy DT: 0.65
Train Accuracy BT: 0.8418803418803419
Test Accuracy BT: 0.6692307692307692
Training Accuracy RF: 0.7948717948717948
Testing Accuracy RF: 0.7038461538461539
Training Accuracy DT: 0.7891737891737892
Test Accuracy DT: 0.6365384615384615
Train Accuracy BT: 0.8233618233618234
Test Accuracy BT: 0.6846153846153846
Training Accuracy RF: 0.792022792022792
Testing Accuracy RF: 0.6846153846153846
Training Accuracy DT: 0.7606837606837606
Test Accuracy DT: 0.6653846153846154
Train Accuracy BT: 0.811965811965812
Test Accuracy BT: 0.6730769230769231
Training Accuracy RF: 0.7628205128205128
Testing Accuracy RF: 0.6884615384615385
Training Accuracy DT: 0.7549857549857549
Test Accuracy DT: 0.7
Train Accuracy BT: 0.8105413105413105
Test Accuracy BT: 0.7076923076923077
Training Accuracy RF: 0.7720797720797721
```

```
Testing Accuracy RF: 0.6846153846153846
Training Accuracy DT: 0.7670940170940171
Test Accuracy DT: 0.6903846153846154
Train Accuracy BT: 0.8098290598290598
Test Accuracy BT: 0.7211538461538461
Training Accuracy RF: 0.7873931623931624
Testing Accuracy RF: 0.6903846153846154
Training Accuracy DT: 0.782051282051282
Test Accuracy DT: 0.7038461538461539
Train Accuracy BT: 0.8247863247863247
Test Accuracy BT: 0.7288461538461538
Training Accuracy RF: 0.8247863247863247
Testing Accuracy RF: 0.7442307692307693
Training Accuracy DT: 0.7777777777777778
Test Accuracy DT: 0.7211538461538461
Train Accuracy BT: 0.8148148148148148
Test Accuracy BT: 0.7442307692307693
Training Accuracy RF: 0.7977207977207977
Testing Accuracy RF: 0.7442307692307693
Training Accuracy DT: 0.7970085470085471
Test Accuracy DT: 0.6903846153846154
Train Accuracy BT: 0.8205128205128205
Test Accuracy BT: 0.7384615384615385
Training Accuracy RF: 0.8012820512820513
Testing Accuracy RF: 0.7326923076923076
Training Accuracy DT: 0.7834757834757835
Test Accuracy DT: 0.676923076923077
Train Accuracy BT: 0.8133903133903134
Test Accuracy BT: 0.7365384615384616
Training Accuracy RF: 0.7678062678062678
Testing Accuracy RF: 0.7519230769230769
Training Accuracy DT: 0.7638888888888888
Test Accuracy DT: 0.7057692307692308
Train Accuracy BT: 0.7970085470085471
Test Accuracy BT: 0.7634615384615384
Training Accuracy RF: 0.7681623931623932
Testing Accuracy RF: 0.75
Training Accuracy DT: 0.7692307692307693
Test Accuracy DT: 0.6615384615384615
Train Accuracy BT: 0.8290598290598291
Test Accuracy BT: 0.6807692307692308
Training Accuracy RF: 0.8333333333333334
Testing Accuracy RF: 0.6961538461538461
Training Accuracy DT: 0.7806267806267806
Test Accuracy DT: 0.6961538461538461
Train Accuracy BT: 0.8376068376068376
Test Accuracy BT: 0.7153846153846154
Training Accuracy RF: 0.7948717948717948
```

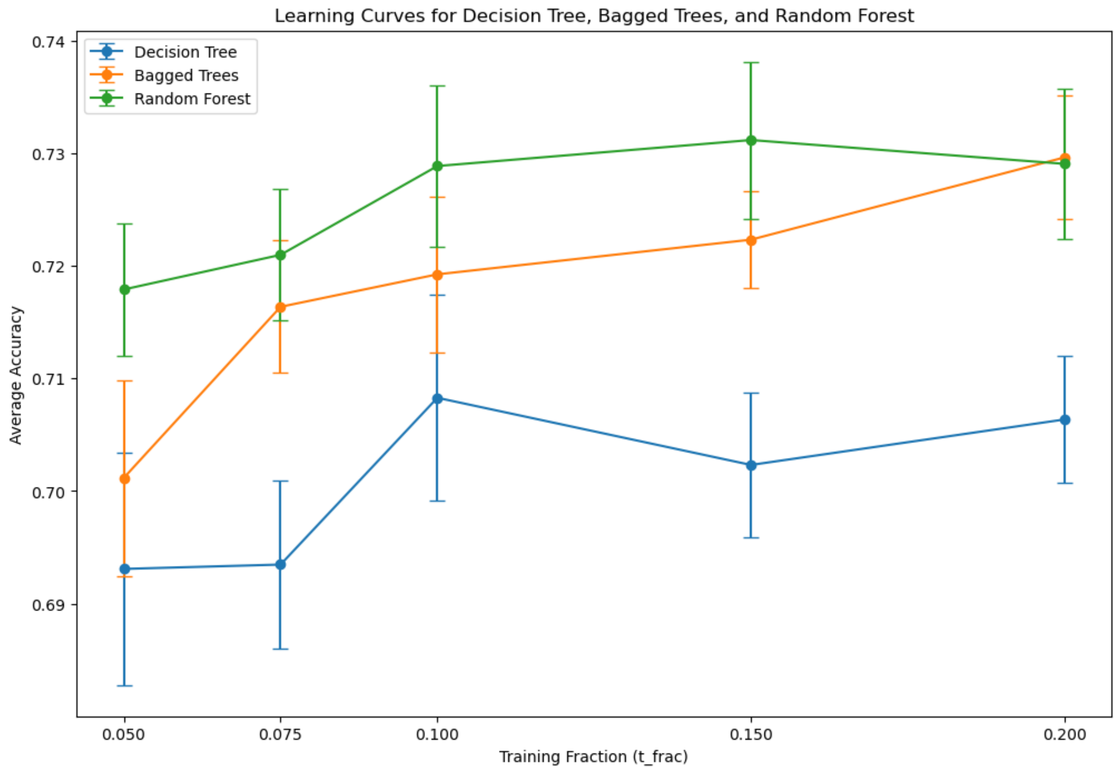
```
Testing Accuracy RF: 0.7076923076923077
Training Accuracy DT: 0.7884615384615384
Test Accuracy DT: 0.6807692307692308
Train Accuracy BT: 0.844017094017094
Test Accuracy BT: 0.7096153846153846
Training Accuracy RF: 0.8034188034188035
Testing Accuracy RF: 0.7173076923076923
Training Accuracy DT: 0.7934472934472935
Test Accuracy DT: 0.6711538461538461
Train Accuracy BT: 0.8247863247863247
Test Accuracy BT: 0.7230769230769231
Training Accuracy RF: 0.782051282051282
Testing Accuracy RF: 0.725
Training Accuracy DT: 0.7702991452991453
Test Accuracy DT: 0.7230769230769231
Train Accuracy BT: 0.811965811965812
Test Accuracy BT: 0.725
Training Accuracy RF: 0.7841880341880342
Testing Accuracy RF: 0.7288461538461538
Training Accuracy DT: 0.782051282051282
Test Accuracy DT: 0.7134615384615385
Train Accuracy BT: 0.8376068376068376
Test Accuracy BT: 0.6884615384615385
Training Accuracy RF: 0.811965811965812
Testing Accuracy RF: 0.698076923076923
Training Accuracy DT: 0.7806267806267806
Test Accuracy DT: 0.675
Train Accuracy BT: 0.8176638176638177
Test Accuracy BT: 0.7288461538461538
Training Accuracy RF: 0.7777777777777778
Testing Accuracy RF: 0.7307692307692307
Training Accuracy DT: 0.7927350427350427
Test Accuracy DT: 0.6826923076923077
Train Accuracy BT: 0.8290598290598291
Test Accuracy BT: 0.7346153846153847
Training Accuracy RF: 0.782051282051282
Testing Accuracy RF: 0.7423076923076923
Training Accuracy DT: 0.7863247863247863
Test Accuracy DT: 0.7096153846153846
Train Accuracy BT: 0.8034188034188035
Test Accuracy BT: 0.7307692307692307
Training Accuracy RF: 0.7905982905982906
Testing Accuracy RF: 0.7365384615384616
Training Accuracy DT: 0.7702991452991453
Test Accuracy DT: 0.7173076923076923
Train Accuracy BT: 0.7970085470085471
Test Accuracy BT: 0.7423076923076923
Training Accuracy RF: 0.7628205128205128
```

```
Testing Accuracy RF: 0.7442307692307693
Training Accuracy DT: 0.7649572649572649
Test Accuracy DT: 0.6923076923076923
Train Accuracy BT: 0.8589743589743589
Test Accuracy BT: 0.7269230769230769
Training Accuracy RF: 0.8247863247863247
Testing Accuracy RF: 0.7288461538461538
Training Accuracy DT: 0.7720797720797721
Test Accuracy DT: 0.6923076923076923
Train Accuracy BT: 0.8233618233618234
Test Accuracy BT: 0.725
Training Accuracy RF: 0.7806267806267806
Testing Accuracy RF: 0.7365384615384616
Training Accuracy DT: 0.7841880341880342
Test Accuracy DT: 0.7230769230769231
Train Accuracy BT: 0.8141025641025641
Test Accuracy BT: 0.7461538461538462
Training Accuracy RF: 0.7948717948717948
Testing Accuracy RF: 0.7673076923076924
Training Accuracy DT: 0.7834757834757835
Test Accuracy DT: 0.7230769230769231
Train Accuracy BT: 0.8133903133903134
Test Accuracy BT: 0.7403846153846154
Training Accuracy RF: 0.7806267806267806
Testing Accuracy RF: 0.7596153846153846
Training Accuracy DT: 0.7649572649572649
Test Accuracy DT: 0.6923076923076923
Train Accuracy BT: 0.7980769230769231
Test Accuracy BT: 0.7288461538461538
Training Accuracy RF: 0.7724358974358975
Testing Accuracy RF: 0.7596153846153846
Training Accuracy DT: 0.7948717948717948
Test Accuracy DT: 0.698076923076923
Train Accuracy BT: 0.811965811965812
Test Accuracy BT: 0.7038461538461539
Training Accuracy RF: 0.8162393162393162
Testing Accuracy RF: 0.7326923076923076
Training Accuracy DT: 0.7806267806267806
Test Accuracy DT: 0.7134615384615385
Train Accuracy BT: 0.8148148148148148
Test Accuracy BT: 0.7192307692307692
Training Accuracy RF: 0.7891737891737892
Testing Accuracy RF: 0.7230769230769231
Training Accuracy DT: 0.782051282051282
Test Accuracy DT: 0.7307692307692307
Train Accuracy BT: 0.8162393162393162
Test Accuracy BT: 0.7288461538461538
Training Accuracy RF: 0.7905982905982906
```

```
Testing Accuracy RF: 0.75
Training Accuracy DT: 0.7777777777777778
Test Accuracy DT: 0.7076923076923077
Train Accuracy BT: 0.8048433048433048
Test Accuracy BT: 0.7019230769230769
Training Accuracy RF: 0.792022792022792
Testing Accuracy RF: 0.7326923076923076
Training Accuracy DT: 0.7692307692307693
Test Accuracy DT: 0.7153846153846154
Train Accuracy BT: 0.8023504273504274
Test Accuracy BT: 0.7192307692307692
Training Accuracy RF: 0.7756410256410257
Testing Accuracy RF: 0.725
Training Accuracy DT: 0.8076923076923077
Test Accuracy DT: 0.6788461538461539
Train Accuracy BT: 0.8333333333333334
Test Accuracy BT: 0.7
Training Accuracy RF: 0.8418803418803419
Testing Accuracy RF: 0.7211538461538461
Training Accuracy DT: 0.7606837606837606
Test Accuracy DT: 0.6884615384615385
Train Accuracy BT: 0.7977207977207977
Test Accuracy BT: 0.6846153846153846
Training Accuracy RF: 0.8148148148148148
Testing Accuracy RF: 0.7173076923076923
Training Accuracy DT: 0.7927350427350427
Test Accuracy DT: 0.7173076923076923
Train Accuracy BT: 0.811965811965812
Test Accuracy BT: 0.698076923076923
Training Accuracy RF: 0.8076923076923077
Testing Accuracy RF: 0.7326923076923076
Training Accuracy DT: 0.7877492877492878
Test Accuracy DT: 0.7057692307692308
Train Accuracy BT: 0.8162393162393162
Test Accuracy BT: 0.7230769230769231
Training Accuracy RF: 0.7934472934472935
Testing Accuracy RF: 0.7153846153846154
Training Accuracy DT: 0.7724358974358975
Test Accuracy DT: 0.7134615384615385
Train Accuracy BT: 0.8087606837606838
Test Accuracy BT: 0.7192307692307692
Training Accuracy RF: 0.7777777777777778
Testing Accuracy RF: 0.7115384615384616
Training Accuracy DT: 0.7905982905982906
Test Accuracy DT: 0.7346153846153847
Train Accuracy BT: 0.8247863247863247
Test Accuracy BT: 0.7288461538461538
Training Accuracy RF: 0.8034188034188035
```

```
Testing Accuracy RF: 0.7365384615384616
Training Accuracy DT: 0.792022792022792
Test Accuracy DT: 0.6865384615384615
Train Accuracy BT: 0.8205128205128205
Test Accuracy BT: 0.7192307692307692
Training Accuracy RF: 0.8034188034188035
Testing Accuracy RF: 0.7423076923076923
Training Accuracy DT: 0.7991452991452992
Test Accuracy DT: 0.7230769230769231
Train Accuracy BT: 0.811965811965812
Test Accuracy BT: 0.7288461538461538
Training Accuracy RF: 0.8205128205128205
Testing Accuracy RF: 0.7365384615384616
Training Accuracy DT: 0.8048433048433048
Test Accuracy DT: 0.6884615384615385
Train Accuracy BT: 0.8162393162393162
Test Accuracy BT: 0.7211538461538461
Training Accuracy RF: 0.8190883190883191
Testing Accuracy RF: 0.7326923076923076
Training Accuracy DT: 0.7895299145299145
Test Accuracy DT: 0.676923076923077
Train Accuracy BT: 0.8141025641025641
Test Accuracy BT: 0.7096153846153846
Training Accuracy RF: 0.7895299145299145
Testing Accuracy RF: 0.7365384615384616
Training Accuracy DT: 0.7905982905982906
Test Accuracy DT: 0.75
Train Accuracy BT: 0.8376068376068376
Test Accuracy BT: 0.7346153846153847
Training Accuracy RF: 0.8162393162393162
Testing Accuracy RF: 0.7288461538461538
Training Accuracy DT: 0.792022792022792
Test Accuracy DT: 0.7153846153846154
Train Accuracy BT: 0.8376068376068376
Test Accuracy BT: 0.7346153846153847
Training Accuracy RF: 0.8205128205128205
Testing Accuracy RF: 0.725
Training Accuracy DT: 0.8034188034188035
Test Accuracy DT: 0.7692307692307693
Train Accuracy BT: 0.8482905982905983
Test Accuracy BT: 0.7346153846153847
Training Accuracy RF: 0.8333333333333334
Testing Accuracy RF: 0.7269230769230769
Training Accuracy DT: 0.7991452991452992
Test Accuracy DT: 0.7442307692307693
Train Accuracy BT: 0.811965811965812
Test Accuracy BT: 0.7365384615384616
Training Accuracy RF: 0.8076923076923077
```

```
Testing Accuracy RF: 0.7596153846153846
Training Accuracy DT: 0.7788461538461539
Test Accuracy DT: 0.7384615384615385
Train Accuracy BT: 0.8034188034188035
Test Accuracy BT: 0.7557692307692307
Training Accuracy RF: 0.7852564102564102
Testing Accuracy RF: 0.7423076923076923
Training Accuracy DT: 0.7991452991452992
Test Accuracy DT: 0.6480769230769231
Train Accuracy BT: 0.7991452991452992
Test Accuracy BT: 0.65
Training Accuracy RF: 0.811965811965812
Testing Accuracy RF: 0.6884615384615385
Training Accuracy DT: 0.7834757834757835
Test Accuracy DT: 0.7096153846153846
Train Accuracy BT: 0.8205128205128205
Test Accuracy BT: 0.7076923076923077
Training Accuracy RF: 0.8262108262108262
Testing Accuracy RF: 0.698076923076923
Training Accuracy DT: 0.8098290598290598
Test Accuracy DT: 0.7
Train Accuracy BT: 0.8247863247863247
Test Accuracy BT: 0.7
Training Accuracy RF: 0.8226495726495726
Testing Accuracy RF: 0.6942307692307692
Training Accuracy DT: 0.8076923076923077
Test Accuracy DT: 0.6961538461538461
Train Accuracy BT: 0.8333333333333334
Test Accuracy BT: 0.7019230769230769
Training Accuracy RF: 0.8105413105413105
Testing Accuracy RF: 0.7134615384615385
Training Accuracy DT: 0.8034188034188035
Test Accuracy DT: 0.6903846153846154
Train Accuracy BT: 0.8087606837606838
Test Accuracy BT: 0.7115384615384616
Training Accuracy RF: 0.7873931623931624
Testing Accuracy RF: 0.7019230769230769
Decision Tree:
  Training fraction 0.05: [0.65, 0.7038461538461539, 0.6615384615384615,
0.7134615384615385, 0.6923076923076923, 0.698076923076923, 0.6788461538461539,
0.7346153846153847, 0.75, 0.6480769230769231]
  Training fraction 0.075: [0.6365384615384615, 0.7211538461538461,
0.6961538461538461, 0.675, 0.6923076923076923, 0.7134615384615385,
0.6884615384615385, 0.6865384615384615, 0.7153846153846154, 0.7096153846153846]
  Training fraction 0.1: [0.6653846153846154, 0.6903846153846154,
0.6807692307692308, 0.6826923076923077, 0.7230769230769231, 0.7307692307692307,
0.7173076923076923, 0.7230769230769231, 0.7692307692307693, 0.7]
  Training fraction 0.15: [0.7, 0.676923076923077, 0.6711538461538461,
0.7096153846153846, 0.7230769230769231, 0.7076923076923077, 0.7057692307692308,
0.6884615384615385, 0.7442307692307693, 0.6961538461538461]
  Training fraction 0.2: [0.6903846153846154, 0.7057692307692308,
0.7230769230769231, 0.7173076923076923, 0.6923076923076923, 0.7153846153846154,
0.7134615384615385, 0.676923076923077, 0.7384615384615385, 0.6903846153846154]
Bagged Trees:
  Training fraction 0.05: [0.6692307692307692, 0.7288461538461538,
0.6807692307692308, 0.6884615384615385, 0.7269230769230769, 0.7038461538461539,
0.7, 0.7288461538461538, 0.7346153846153847, 0.65]
  Training fraction 0.075: [0.6846153846153846, 0.7442307692307693,
0.7153846153846154, 0.7288461538461538, 0.725, 0.7192307692307692,
0.6846153846153846, 0.7192307692307692, 0.7346153846153847, 0.7076923076923077]
  Training fraction 0.1: [0.6730769230769231, 0.7384615384615385,
0.7096153846153846, 0.7346153846153847, 0.7461538461538462, 0.7288461538461538,
0.698076923076923, 0.7288461538461538, 0.7346153846153847, 0.7]
  Training fraction 0.15: [0.7076923076923077, 0.7365384615384616,
0.7230769230769231, 0.7307692307692307, 0.7403846153846154, 0.7019230769230769,
0.7230769230769231, 0.7211538461538461, 0.7365384615384616, 0.7019230769230769]
  Training fraction 0.2: [0.7211538461538461, 0.7634615384615384, 0.725,
0.7423076923076923, 0.7288461538461538, 0.7192307692307692, 0.7192307692307692,
0.7096153846153846, 0.7557692307692307, 0.7115384615384616]
Random Forest:
  Training fraction 0.05: [0.7038461538461539, 0.7442307692307693,
0.6961538461538461, 0.698076923076923, 0.7288461538461538, 0.7326923076923076,
0.7211538461538461, 0.7365384615384616, 0.7288461538461538, 0.6884615384615385]
  Training fraction 0.075: [0.6846153846153846, 0.7442307692307693,
0.7076923076923077, 0.7307692307692307, 0.7365384615384616, 0.7230769230769231,
0.7173076923076923, 0.7423076923076923, 0.725, 0.698076923076923]
  Training fraction 0.1: [0.6884615384615385, 0.7326923076923076,
0.7173076923076923, 0.7423076923076923, 0.7673076923076924, 0.75,
0.7326923076923076, 0.7365384615384616, 0.7269230769230769, 0.6942307692307692]
  Training fraction 0.15: [0.6846153846153846, 0.7519230769230769, 0.725,
0.7365384615384616, 0.7596153846153846, 0.7326923076923076, 0.7153846153846154,
0.7326923076923076, 0.7596153846153846, 0.7134615384615385]
  Training fraction 0.2: [0.6903846153846154, 0.75, 0.7288461538461538,
0.7442307692307693, 0.7596153846153846, 0.725, 0.7115384615384616,
0.7365384615384616, 0.7423076923076923, 0.7019230769230769]
```



1.6 4) b) Hypothesis

Null Hypothesis (H0): There is no significant difference in performance (accuracy) between the decision tree and the random forest. Alternative Hypothesis (H1): The random forest has significantly higher accuracy than the decision tree.

[31]:
```
from scipy import stats


fractions = [0.05, 0.075, 0.1, 0.15, 0.2]
alpha = 0.05

for frac in fractions:
    dt_accuracies = dict_for_models_acc['Decision Tree'][frac]
    rf_accuracies = dict_for_models_acc['Random Forest'][frac]
    t_stat, p_value = stats.ttest_ind(dt_accuracies, rf_accuracies)
    if p_value < alpha:
        print(f"For Training Fraction {frac}: p-value = {p_value} - Reject the␣
 ↪Null Hypothesis")
    else:
        print(f"For Training Fraction {frac}: p-value = {p_value} - Fail to␣
↪Reject the Null Hypothesis")
```

```
For Training Fraction 0.05: p-value = 0.062456711433191335 - Fail to Reject the
Null Hypothesis
For Training Fraction 0.075: p-value = 0.013009242228505169 - Reject the Null
Hypothesis
For Training Fraction 0.1: p-value = 0.1107407622884971 - Fail to Reject the
Null Hypothesis
For Training Fraction 0.15: p-value = 0.009812265868715928 - Reject the Null
Hypothesis
For Training Fraction 0.2: p-value = 0.02317972000117481 - Reject the Null
Hypothesis
```

We have chosen an alpha value of 0.05. And we reject the null hypothesis if the p-value is less than alpha. This means for t_frac 0.05 and 0.1, we accept the null hypothesis since p-value is more than 0.05 for this case. In all the other t_fracs, we reject the null hypothesis since the p-value is less than 0.05. We can also see this from the graph. *To incorporate specific training set size we have run the cross validation methods and used the accuracy for each t_frac

---

1.7 5) a) The Influence of Number of Trees on Classifier Performance (10 points)

[33]:
```
import pandas as pd
import numpy as np
from statistics import mode, multimode
from scipy.stats import mode

def trees(trainingDataFilename, testDataFilename, modelIdx, num_trees = 30):
    def decisionTree(trainingSet, testSet):
        def gini_full(y):
            m = len(y)
            y_list = list(y)
            unique_elements = np.unique(y)
            sum_of_squares = 0
            for element in unique_elements:
                count = y_list.count(element)
                proportion = count / m
                squared_proportion = proportion ** 2
                sum_of_squares += squared_proportion
            gini_impurity = 1.0 - sum_of_squares
            return gini_impurity

        def split_score_current_best_split(X, y):
            split_score_current_best = float('inf')
            split_score_current_best_split = None
            bestY, bestX = None, None
            counter = 0
            columns = X.columns.tolist()
            while counter < len(columns):
                column = columns[counter]
                all_possible_t = X[column].unique()
                t_val_index = 0
                while t_val_index < len(all_possible_t):
                    t_val = all_possible_t[t_val_index]
                    left_y = y[X[column] < t_val]
                    right_y = y[X[column] >= t_val]
                    left_gini = gini_full(left_y)
                    right_gini = gini_full(right_y)
                    score = (len(left_y) / len(y)) * left_gini + (len(right_y) /
    ↪ len(y)) * right_gini
                    if score < split_score_current_best:
                        split_score_current_best = score
                        split_score_current_best_split = (column, t_val)
                        bestY, bestX = left_y, right_y
                    t_val_index += 1
                counter += 1
            return split_score_current_best, split_score_current_best_split,␣
    ↪bestY, bestX

        def build_tree(X, y, depth=0):
            conditions = {
                1: lambda: depth == 8,
                2: lambda: len(y) < 50,
                3: lambda: len(np.unique(y)) == 1
            }
            for key, condition in conditions.items():
                if condition():
                    return {'node': {'prediction': y.mode()[0]}}

            _, (column, t_val), left_y, right_y =␣
    ↪split_score_current_best_split(X, y)
            if not column:
                return {'node': {'prediction': y.mode()[0]}}

            left_tree = build_tree(X[X[column] < t_val], left_y, depth + 1)
            right_tree = build_tree(X[X[column] >= t_val], right_y, depth + 1)

            return {
                'decision': {
                    'column': column,
                    'threshold': t_val
                },
                'branches': {
                    'left': left_tree,
                    'right': right_tree
                }
            }

        def predict(tree, x):
            while True:
                if 'node' in tree:
                    return tree['node']['prediction']
                column, t_val = tree['decision']['column'],␣
↪tree['decision']['threshold']
                tree = tree['branches']['left'] if x[column] < t_val else␣
↪tree['branches']['right']

        train_data = pd.read_csv(trainingSet)
        test_data = pd.read_csv(testSet)
        X_train, y_train = train_data.drop('decision', axis=1),␣
↪train_data['decision']
        X_test, y_test = test_data.drop('decision', axis=1),␣
↪test_data['decision']
        tree = build_tree(X_train, y_train)
        train_predictions = X_train.apply(lambda x: predict(tree, x), axis=1)
        train_accuracy = (train_predictions == y_train).mean()
        test_predictions = X_test.apply(lambda x: predict(tree, x), axis=1)
        test_accuracy = (test_predictions == y_test).mean()
        print(f'Training Accuracy DT: {train_accuracy}')
        print(f'Test Accuracy DT: {test_accuracy}')
        return train_accuracy, test_accuracy

    import pandas as pd
    import numpy as np
    from scipy.stats import mode as scipy_mode

    def bagging(trainingSet, testSet, num_trees):
        def calculate_gini_impurity(y):
            total_items = len(y)
            y_list = list(y)
            unique_elements = np.unique(y)
            impurity_sum = 0
            for element in unique_elements:
                element_count = y_list.count(element)
                element_proportion = element_count / total_items
                impurity_sum += element_proportion ** 2
            return 1.0 - impurity_sum

        def find_optimal_split(X, y):
            best_score_bag = float('inf')
            best_split_bag = None
            bestYbag, bestXbag = None, None

            col_index = 0
            while col_index < len(X.columns):
                column = X.columns[col_index]
                total_t = X[column].unique()
                t_count = 0
                while t_count < len(total_t):
                    current_t = total_t[t_count]
                    left_y, right_y = y[X[column] < current_t], y[X[column] >=␣
    ↪current_t]
                    gini_left, gini_right = calculate_gini_impurity(left_y),␣
    ↪calculate_gini_impurity(right_y)
                    combined_score = (len(left_y) / len(y)) * gini_left +␣
    ↪(len(right_y) / len(y)) * gini_right
                    if combined_score < best_score_bag:
                        best_score_bag = combined_score
                        best_split_bag = (column, current_t)
                        bestYbag, bestXbag = left_y, right_y
                    t_count += 1
                col_index += 1
            return best_score_bag, best_split_bag, bestYbag, bestXbag

        def create_decision_tree(X, y, depth=0):
            if depth == 8 or len(y) < 50 or len(np.unique(y)) == 1:
                return {'node': {'prediction': scipy_mode(y).mode[0]}}

            _, (split_column, split_value), left_y, right_y =␣
    ↪find_optimal_split(X, y)
            if not split_column:
                return {'node': {'prediction': scipy_mode(y).mode[0]}}

            left_branch = create_decision_tree(X[X[split_column] <␣
    ↪split_value], left_y, depth + 1)
            right_branch = create_decision_tree(X[X[split_column] >=␣
    ↪split_value], right_y, depth + 1)

            return {
                'decision_criteria': {
                    'column': split_column,
                    'current_t': split_value
                },
                'subtrees': {
                    'left': left_branch,
                    'right': right_branch
                }
            }

        def tree_prediction(tree, x):
            while 'node' not in tree:
                when_to_split = tree['decision_criteria']
                tree = tree['subtrees']['left'] if x[when_to_split['column']] <␣
↪when_to_split['current_t'] else tree['subtrees']['right']
            return tree['node']['prediction']

        def generate_bootstrap_sample(X, y):
            sample_indices = np.random.randint(0, len(X), len(X))
            return X.iloc[sample_indices], y.iloc[sample_indices]

        def ensemble_prediction(trees, x):
            predictions = [tree_prediction(tree, x) for tree in trees]
            return scipy_mode(predictions).mode[0]

        train_dataset = pd.read_csv(trainingSet)
        test_dataset = pd.read_csv(testSet)
        X_train, y_train = train_dataset.drop('decision', axis=1),␣
↪train_dataset['decision']
        X_test, y_test = test_dataset.drop('decision', axis=1),␣
↪test_dataset['decision']
        decision_trees = []
        tree_index = 0
        while tree_index < num_trees:
            X_sample, y_sample = generate_bootstrap_sample(X_train, y_train)
            tree = create_decision_tree(X_sample, y_sample)
            decision_trees.append(tree)
            tree_index += 1
        train_predictions = X_train.apply(lambda x:␣
↪ensemble_prediction(decision_trees, x), axis=1)
        test_predictions = X_test.apply(lambda x:␣
↪ensemble_prediction(decision_trees, x), axis=1)
        accuracy_train = np.mean(train_predictions == y_train)
        accuracy_test = np.mean(test_predictions == y_test)
        print(f'Train Accuracy BT: {accuracy_train}')
        print(f'Test Accuracy BT: {accuracy_test}')
        return accuracy_train, accuracy_test

    import pandas as pd
    import numpy as np
    from statistics import mode

    def randomForests(trainingSet, testSet, num_trees):
        max_tree_depth = 8
        min_samples = 50

        def calculate_gini_impurity(y):
            total_count = len(y)
            y_list = list(y)
            impurity_sum = 0
            for element in np.unique(y):
                element_count = y_list.count(element)
                element_proportion = element_count / total_count
                impurity_sum += element_proportion ** 2
            return 1.0 - impurity_sum

        def optimal_split(X, y):
            best_score_forest = float('inf')
            best_score_forest_deet = None
            forestY, forestX = None, None

            featForest = np.random.choice(X.columns, int(np.sqrt(len(X.
↪columns))), replace=False)
            for feature in featForest:
                for threshold in X[feature].unique():
                    left, right = y[X[feature] < threshold], y[X[feature] >=␣
↪threshold]
                    curr_score = (len(left) / len(y)) *␣
↪calculate_gini_impurity(left) + (len(right) / len(y)) *␣
↪calculate_gini_impurity(right)
                    if curr_score < best_score_forest:
                        best_score_forest = curr_score
                        best_score_forest_deet = (feature, threshold)
                        forestY, forestX = left, right

            return best_score_forest, best_score_forest_deet, forestY, forestX

        def generate_tree(X, y, depth=0, max_depth=max_tree_depth,␣
↪min_split=min_samples):
            if len(y) == 0:
                return {'node': {'prediction': None}}

            if len(np.unique(y)) == 1 or depth == max_depth or len(y) <␣
↪min_split:
                return {'node': {'prediction': mode(y)}}

            _, (split_feature, split_value), left_y, right_y = optimal_split(X,␣
↪y)
            if not split_feature:
                return {'node': {'prediction': mode(y)}}
            left_branch = generate_tree(X[X[split_feature] < split_value],␣
↪left_y, depth + 1)
            right_branch = generate_tree(X[X[split_feature] >= split_value],␣
↪right_y, depth + 1)

            return {
                'when_to_split': {
                    'feature': split_feature,
                    'value': split_value
                },
                'branches': {
                    'left': left_branch,
                    'right': right_branch
                }
            }


        def predict_from_tree(tree, instance):
            while 'node' not in tree:
                criteria = tree['when_to_split']
                tree = tree['branches']['left'] if␣
↪instance[criteria['feature']] < criteria['value'] else␣
↪tree['branches']['right']
            return tree['node']['prediction']

        #WITH REPLACEMENT, hence setting to TRUE
        def create_forest(X, y, trees_count):
            forest = []
            for _ in range(trees_count):
                sample_Forest = np.random.choice(X.index, size=len(X),␣
↪replace=True)
                X_sample, y_sample = X.loc[sample_Forest], y.loc[sample_Forest]
                tree = generate_tree(X_sample, y_sample)
                forest.append(tree)
            return forest

        def predict_with_forest(forest, X):
            predictions = [[predict_from_tree(tree, x) for tree in forest] for␣
↪_, x in X.iterrows()]
            mode_predictions = [mode(pred) for pred in predictions]
            return mode_predictions

        train_dataset = pd.read_csv(trainingSet)
        test_dataset = pd.read_csv(testSet)
        X_train, y_train = train_dataset.drop('decision', axis=1),␣
↪train_dataset['decision']
```

```
        X_test, y_test = test_dataset.drop('decision', axis=1),␣
↪test_dataset['decision']
        forest = create_forest(X_train, y_train, num_trees)
        train_accuracy = np.mean(predict_with_forest(forest, X_train) ==␣
↪y_train)
        test_accuracy = np.mean(predict_with_forest(forest, X_test) == y_test)
        print(f'Training Accuracy RF: {train_accuracy}')
        print(f'Testing Accuracy RF: {test_accuracy}')

        return train_accuracy, test_accuracy

    if modelIdx == 1:
        return decisionTree(trainingDataFilename, testDataFilename)
    elif modelIdx == 2:
        return bagging(trainingDataFilename, testDataFilename, num_trees)
    elif modelIdx == 3:
        return randomForests(trainingDataFilename, testDataFilename,num_trees)
    else:
        raise ValueError("Please man put the right number.")
```

[34]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

data = pd.read_csv('trainingSet.csv')
shuffled_data = data.sample(frac=1, random_state=18)
sampled_data = shuffled_data.sample(frac=0.5, random_state=32)
folds = np.array_split(sampled_data, 10)
tree_counts = [10, 20, 40, 50]
max_depth = 8
min_samples_split = 50

def manual_cv_scores_custom(trees_function, modelIdx, n_trees, folds):
    scores = []
    for i in range(len(folds)):
        test_data = folds[i]
        train_data = pd.concat([folds[j] for j in range(len(folds)) if j != i])
        train_data.to_csv('train_temp.csv', index=False)
        test_data.to_csv('test_temp.csv', index=False)
        _, test_accuracy = trees_function('train_temp.csv', 'test_temp.csv',␣
↪modelIdx, n_trees)
        scores.append(test_accuracy)
    return np.mean(scores), np.std(scores)

results_custom = {'Bagged Trees': {'mean': [], 'std': []},
                  'Random Forest': {'mean': [], 'std': []}}
for t in tree_counts:
    mean_score, std_score = manual_cv_scores_custom(trees, 2, t, folds)
    results_custom['Bagged Trees']['mean'].append(mean_score)
    results_custom['Bagged Trees']['std'].append(std_score)
    mean_score, std_score = manual_cv_scores_custom(trees, 3, t, folds)
    results_custom['Random Forest']['mean'].append(mean_score)
    results_custom['Random Forest']['std'].append(std_score)
```

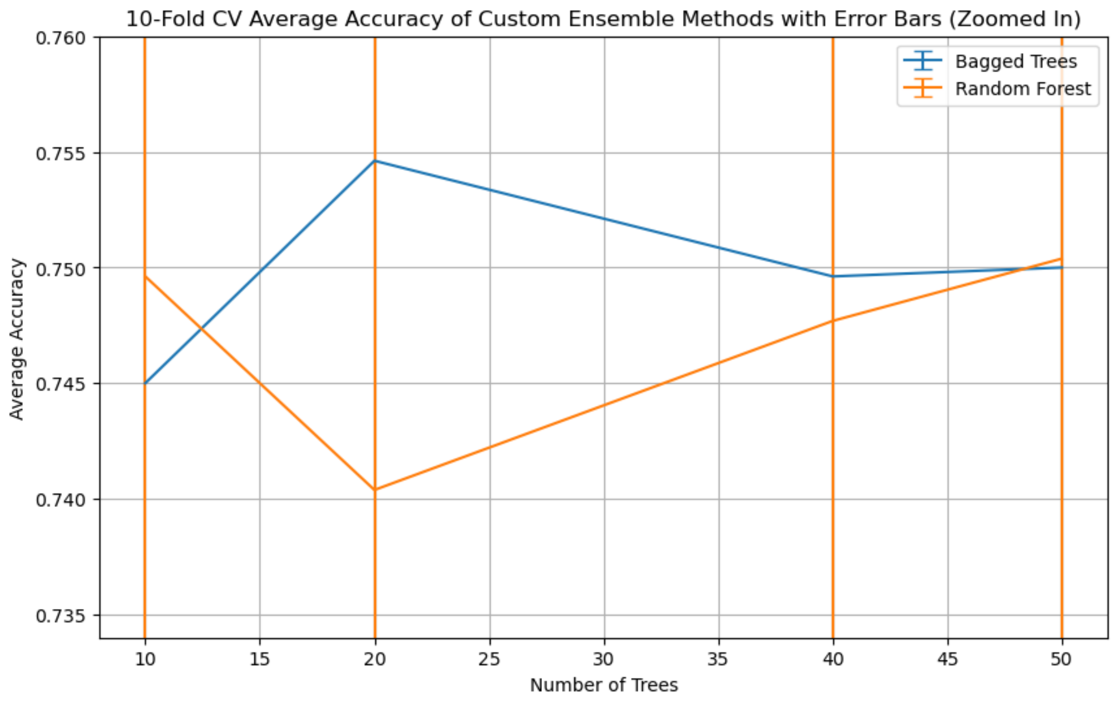
```
Train Accuracy BT: 0.8051282051282052
Test Accuracy BT: 0.7230769230769231
Train Accuracy BT: 0.7991452991452992
Test Accuracy BT: 0.7653846153846153
Train Accuracy BT: 0.7854700854700855
Test Accuracy BT: 0.7615384615384615
Train Accuracy BT: 0.7927350427350427
Test Accuracy BT: 0.7076923076923077
Train Accuracy BT: 0.7987179487179488
Test Accuracy BT: 0.7538461538461538
Train Accuracy BT: 0.791025641025641
Test Accuracy BT: 0.7346153846153847
Train Accuracy BT: 0.7884615384615384
Test Accuracy BT: 0.7423076923076923
Train Accuracy BT: 0.7918803418803418
Test Accuracy BT: 0.7846153846153846
Train Accuracy BT: 0.791025641025641
Test Accuracy BT: 0.7461538461538462
Train Accuracy BT: 0.7893162393162393
Test Accuracy BT: 0.7307692307692307
Training Accuracy RF: 0.7666666666666667
Testing Accuracy RF: 0.7692307692307693
Training Accuracy RF: 0.7811965811965812
Testing Accuracy RF: 0.8038461538461539
Training Accuracy RF: 0.7696581196581197
Testing Accuracy RF: 0.7384615384615385
Training Accuracy RF: 0.7696581196581197
Testing Accuracy RF: 0.6884615384615385
Training Accuracy RF: 0.7653846153846153
Testing Accuracy RF: 0.7615384615384615
Training Accuracy RF: 0.7683760683760684
Testing Accuracy RF: 0.7423076923076923
Training Accuracy RF: 0.7581196581196581
Testing Accuracy RF: 0.7461538461538462
Training Accuracy RF: 0.7645299145299145
Testing Accuracy RF: 0.7576923076923077
Training Accuracy RF: 0.7666666666666667
Testing Accuracy RF: 0.7461538461538462
Training Accuracy RF: 0.7611111111111111
Testing Accuracy RF: 0.7423076923076923
```

```
Train Accuracy BT: 0.7923076923076923
Test Accuracy BT: 0.7346153846153847
Train Accuracy BT: 0.8064102564102564
Test Accuracy BT: 0.7884615384615384
Train Accuracy BT: 0.7991452991452992
Test Accuracy BT: 0.7423076923076923
Train Accuracy BT: 0.7987179487179488
Test Accuracy BT: 0.7269230769230769
Train Accuracy BT: 0.7935897435897435
Test Accuracy BT: 0.7846153846153846
Train Accuracy BT: 0.7957264957264957
Test Accuracy BT: 0.7384615384615385
Train Accuracy BT: 0.8042735042735043
Test Accuracy BT: 0.7346153846153847
Train Accuracy BT: 0.7957264957264957
Test Accuracy BT: 0.8038461538461539
Train Accuracy BT: 0.7961538461538461
Test Accuracy BT: 0.75
Train Accuracy BT: 0.8076923076923077
Test Accuracy BT: 0.7423076923076923
Training Accuracy RF: 0.7769230769230769
Testing Accuracy RF: 0.7576923076923077
Training Accuracy RF: 0.7824786324786325
Testing Accuracy RF: 0.7692307692307693
Training Accuracy RF: 0.7782051282051282
Testing Accuracy RF: 0.7307692307692307
Training Accuracy RF: 0.7743589743589744
Testing Accuracy RF: 0.7
Training Accuracy RF: 0.7581196581196581
Testing Accuracy RF: 0.7615384615384615
Training Accuracy RF: 0.7760683760683761
Testing Accuracy RF: 0.7576923076923077
Training Accuracy RF: 0.7782051282051282
Testing Accuracy RF: 0.7115384615384616
Training Accuracy RF: 0.7623931623931623
Testing Accuracy RF: 0.7538461538461538
Training Accuracy RF: 0.7794871794871795
Testing Accuracy RF: 0.7346153846153847
Training Accuracy RF: 0.7747863247863248
Testing Accuracy RF: 0.7269230769230769
Train Accuracy BT: 0.7944444444444444
Test Accuracy BT: 0.7423076923076923
Train Accuracy BT: 0.7995726495726496
Test Accuracy BT: 0.7653846153846153
Train Accuracy BT: 0.805982905982906
Test Accuracy BT: 0.7692307692307693
Train Accuracy BT: 0.7982905982905983
Test Accuracy BT: 0.7384615384615385
```

```
Train Accuracy BT: 0.7957264957264957
Test Accuracy BT: 0.7538461538461538
Train Accuracy BT: 0.7978632478632479
Test Accuracy BT: 0.7346153846153847
Train Accuracy BT: 0.7995726495726496
Test Accuracy BT: 0.7461538461538462
Train Accuracy BT: 0.7931623931623931
Test Accuracy BT: 0.7961538461538461
Train Accuracy BT: 0.7914529914529914
Test Accuracy BT: 0.7346153846153847
Train Accuracy BT: 0.805982905982906
Test Accuracy BT: 0.7153846153846154
Training Accuracy RF: 0.7726495726495727
Testing Accuracy RF: 0.7423076923076923
Training Accuracy RF: 0.7662393162393163
Testing Accuracy RF: 0.7692307692307693
Training Accuracy RF: 0.7803418803418803
Testing Accuracy RF: 0.7461538461538462
Training Accuracy RF: 0.7717948717948718
Testing Accuracy RF: 0.7153846153846154
Training Accuracy RF: 0.7807692307692308
Testing Accuracy RF: 0.7769230769230769
Training Accuracy RF: 0.7773504273504274
Testing Accuracy RF: 0.7576923076923077
Training Accuracy RF: 0.7730769230769231
Testing Accuracy RF: 0.7307692307692307
Training Accuracy RF: 0.7636752136752136
Testing Accuracy RF: 0.7692307692307693
Training Accuracy RF: 0.7794871794871795
Testing Accuracy RF: 0.7384615384615385
Training Accuracy RF: 0.7688034188034188
Testing Accuracy RF: 0.7307692307692307
Train Accuracy BT: 0.7935897435897435
Test Accuracy BT: 0.75
Train Accuracy BT: 0.8021367521367522
Test Accuracy BT: 0.7692307692307693
Train Accuracy BT: 0.8021367521367522
Test Accuracy BT: 0.7615384615384615
Train Accuracy BT: 0.8042735042735043
Test Accuracy BT: 0.7230769230769231
Train Accuracy BT: 0.7923076923076923
Test Accuracy BT: 0.7576923076923077
Train Accuracy BT: 0.8021367521367522
Test Accuracy BT: 0.75
Train Accuracy BT: 0.8012820512820513
Test Accuracy BT: 0.7461538461538462
Train Accuracy BT: 0.7995726495726496
Test Accuracy BT: 0.7807692307692308
```

```
Train Accuracy BT: 0.7961538461538461
Test Accuracy BT: 0.7423076923076923
Train Accuracy BT: 0.8047008547008547
Test Accuracy BT: 0.7192307692307692
Training Accuracy RF: 0.7824786324786325
Testing Accuracy RF: 0.7423076923076923
Training Accuracy RF: 0.7905982905982906
Testing Accuracy RF: 0.7730769230769231
Training Accuracy RF: 0.7726495726495727
Testing Accuracy RF: 0.7461538461538462
Training Accuracy RF: 0.7760683760683761
Testing Accuracy RF: 0.7192307692307692
Training Accuracy RF: 0.7722222222222223
Testing Accuracy RF: 0.7961538461538461
Training Accuracy RF: 0.7752136752136752
Testing Accuracy RF: 0.7461538461538462
Training Accuracy RF: 0.7803418803418803
Testing Accuracy RF: 0.7269230769230769
Training Accuracy RF: 0.7752136752136752
Testing Accuracy RF: 0.7884615384615384
Training Accuracy RF: 0.7726495726495727
Testing Accuracy RF: 0.7384615384615385
Training Accuracy RF: 0.7760683760683761
Testing Accuracy RF: 0.7269230769230769
```

[40]:
```
plt.figure(figsize=(10, 6))
for model, scores in results_custom.items():
    plt.errorbar(tree_counts, scores['mean'], yerr=scores['std'], capsize=5,␣
 ↪label=model)

plt.xlabel('Number of Trees')
plt.ylabel('Average Accuracy')
plt.title('10-Fold CV Average Accuracy of Custom Ensemble Methods with Error␣
 ↪Bars (Zoomed In)')
plt.legend()
plt.grid(True)
plt.ylim(0.734, 0.76)
plt.show()
```



1.8 5) b) Hypothesis

Null Hypothesis (H0): There is no significant difference in performance of BT and RF Alternative Hypothesis (H1): There is a significant difference in performance of BT and RF

[43]:
```
#2 tailed test
from scipy import stats
tree_counts = [10, 20, 40, 50]
mean_accuracies_bt = np.array(results_custom['Bagged Trees']['mean'])
mean_accuracies_rf = np.array(results_custom['Random Forest']['mean'])
std_errors_bt = np.array(results_custom['Bagged Trees']['std']) / np.sqrt(10)
std_errors_rf = np.array(results_custom['Random Forest']['std']) / np.sqrt(10)
for i, t in enumerate(tree_counts):
    combined_se = np.sqrt(std_errors_bt[i]**2 + std_errors_rf[i]**2)
    t_statistic = (mean_accuracies_bt[i] - mean_accuracies_rf[i]) / combined_se
    df = (2 * (10 - 1))
    p_value = stats.t.sf(np.abs(t_statistic), df=df) * 2
    print(f'Tree count {t}: t-statistic = {t_statistic}, p-value = {p_value}')
    if p_value < 0.05:
        print(f'At tree count {t}, there is a significant difference in␣
↪performance (p < 0.05).')
    else:
        print(f'At tree count {t}, there is no significant difference in␣
↪performance (p >= 0.05).')
```

```
Tree count 10: t-statistic = -0.42034166646243193, p-value = 0.6792111878344022
At tree count 10, there is no significant difference in performance (p >= 0.05).
Tree count 20: t-statistic = 1.3298516312222473, p-value = 0.20017415635384223
At tree count 20, there is no significant difference in performance (p >= 0.05).
Tree count 40: t-statistic = 0.21179976934170416, p-value = 0.8346429541284579
At tree count 40, there is no significant difference in performance (p >= 0.05).
Tree count 50: t-statistic = -0.03919610065230222, p-value = 0.9691655090171745
At tree count 50, there is no significant difference in performance (p >= 0.05).
```

We have chosen an alpha value of 0.05. And we reject the null hypothesis if the p-value is less than alpha. This means for all the tree counts, we accept the null hypothesis since p-value is more than 0.05 for this case. . We can also see this from the graph. *To incorporate specific training set size we have run the cross validation methods and used the accuracy for each tree count

---

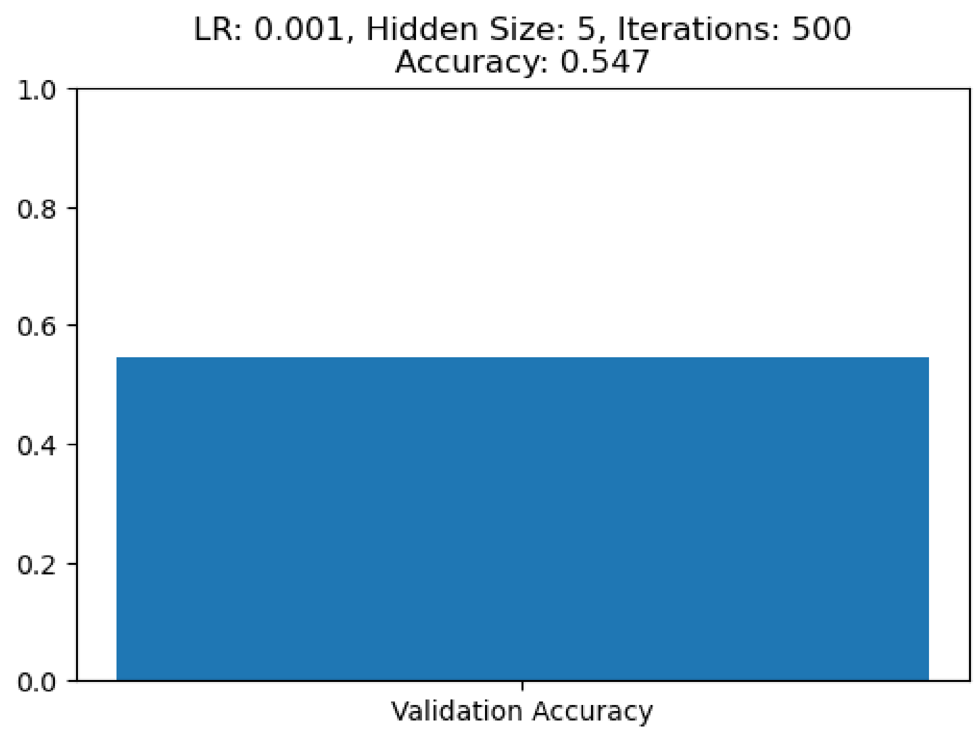
1.9 Bonus question (5 points)

[56]:
```
import numpy as np
import pandas as pd

def initialize_parameters(intake_bounus_q, hidden_bounus_q, output_size):
    W1 = np.random.randn(intake_bounus_q, hidden_bounus_q) * np.sqrt(2. /␣
↪intake_bounus_q)
    b1 = np.zeros((1, hidden_bounus_q))
    W2 = np.random.randn(hidden_bounus_q, output_size) * np.sqrt(2. /␣
↪hidden_bounus_q)
    b2 = np.zeros((1, output_size))
    return W1, b1, W2, b2

def relu(Z):
    return np.maximum(0, Z)

def softmax(Z):
    expZ = np.exp(Z - np.max(Z, axis=1, keepdims=True))
    return expZ / np.sum(expZ, axis=1, keepdims=True)

def sigmoid(Z):
    return 1 / (1 + np.exp(-Z))

def compute_loss_men(Y, A2):
    m = Y.shape[0]
    loss = -np.sum(Y * np.log(A2) + (1 - Y) * np.log(1 - A2)) / m
    return loss
```

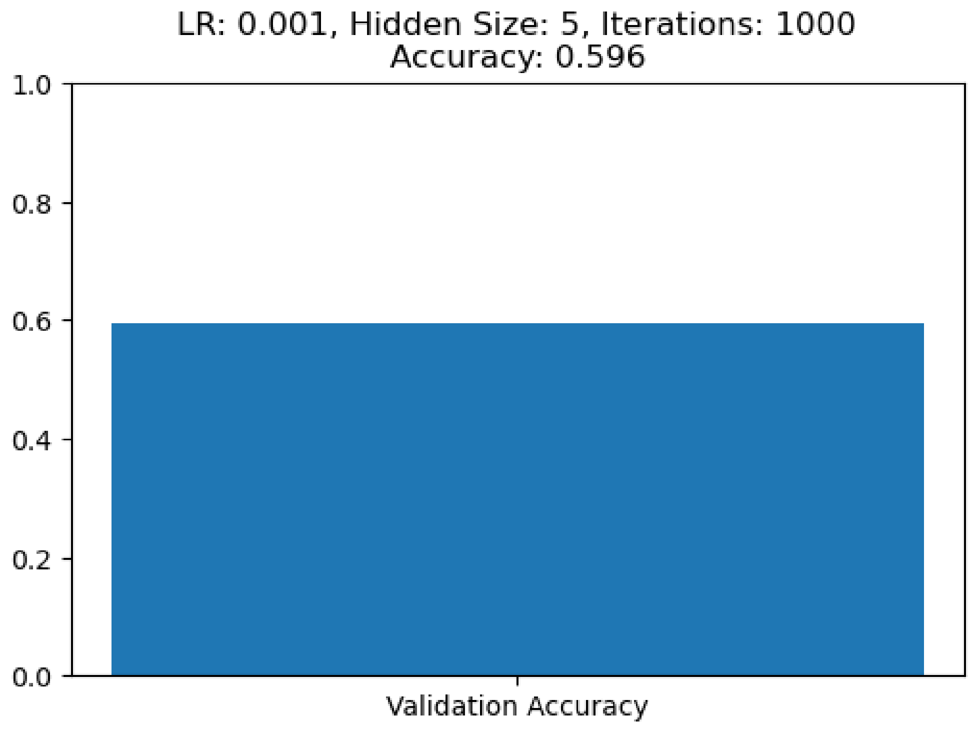
```
def forward_propagation(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = relu(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = sigmoid(Z2)
    return Z1, A1, Z2, A2

def backward_propagation(X, Y, Z1, A1, Z2, A2, W2):
    m = X.shape[0]
    dZ2 = A2 - Y
    dW2 = np.dot(A1.T, dZ2) / m
    db2 = np.sum(dZ2, axis=0, keepdims=True) / m
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = np.dot(X.T, dZ1) / m
    db1 = np.sum(dZ1, axis=0, keepdims=True) / m
    return dW1, db1, dW2, db2

def update_parameters_for_bonus(W1, b1, W2, b2, dW1, db1, dW2, db2,␣
 ↪learning_rate):
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    return W1, b1, W2, b2

def train(X, Y, intake_bounus_q, hidden_bounus_q, output_size, learning_rate,␣
 ↪num_iterations):
    W1, b1, W2, b2 = initialize_parameters(intake_bounus_q, hidden_bounus_q,␣
 ↪output_size)
    for i in range(num_iterations):
        Z1, A1, Z2, A2 = forward_propagation(X, W1, b1, W2, b2)
        loss = compute_loss_men(Y, A2)
        dW1, db1, dW2, db2 = backward_propagation(X, Y, Z1, A1, Z2, A2, W2)
        W1, b1, W2, b2 = update_parameters_for_bonus(W1, b1, W2, b2, dW1, db1,␣
 ↪dW2, db2, learning_rate)
        if i % 100 == 0:
            print(f"Iteration {i} loss: {loss}")

    return W1, b1, W2, b2


def predict(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_propagation(X, W1, b1, W2, b2)
    predictions = A2 > 0.5
    return predictions
```

```
def accuracy(y_true, y_pred):
    correct_predictions = np.sum(y_true == y_pred)
    total_predictions = y_true.shape[0]
    return correct_predictions / total_predictions

def manual_split_dataset(X, y, test_size=0.2, val_size=0.2):
    total_size = len(X)
    test_split_idx = int(total_size * (1 - test_size))
    val_split_idx = int(test_split_idx * (1 - val_size))
    indices = np.random.permutation(total_size)
    train_idx, val_idx, test_idx = indices[:val_split_idx],␣
 ↪indices[val_split_idx:test_split_idx], indices[test_split_idx:]
    X_train, X_val, X_test = X[train_idx], X[val_idx], X[test_idx]
    y_train, y_val, y_test = y[train_idx], y[val_idx], y[test_idx]
    return X_train, X_val, X_test, y_train, y_val, y_test

def manual_standard_scale(X):
    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)
    return (X - mean) / std

train_df = pd.read_csv('trainingSet.csv')
test_df = pd.read_csv('testSet.csv')
X_train = train_df.iloc[:, :-1].values
y_train = train_df.iloc[:, -1].values.reshape(-1, 1)
X_test = test_df.iloc[:, :-1].values
y_test = test_df.iloc[:, -1].values.reshape(-1, 1)
X_train_scaled, X_val_scaled, _, y_train, y_val, _ =␣
 ↪manual_split_dataset(X_train, y_train, test_size=0.4, val_size=0.5)
X_train_val_scaled = np.vstack((X_train_scaled, X_val_scaled))
y_train_val = np.vstack((y_train, y_val))
X_test_scaled = manual_standard_scale(X_test)
intake_bounus_q = X_train_val_scaled.shape[1]
output_size = 1  # Fixed for binary classification
learning_rates = [0.001, 0.01, 0.1]
hidden_sizes = [5, 10, 20]
num_iterations_values = [500, 1000, 1500]
grid_accuracy = np.zeros((len(learning_rates), len(hidden_sizes),␣
 ↪len(num_iterations_values)))
best_accuracy = 0
best_lr = 0
best_hidden_size = 0
best_num_iterations = 0
for i, lr in enumerate(learning_rates):
    for j, hidden_size in enumerate(hidden_sizes):
        for k, num_iterations in enumerate(num_iterations_values):
            W1, b1, W2, b2 = train(X_train_scaled, y_train, intake_bounus_q,␣
 ↪hidden_size, output_size, lr, num_iterations)
            y_pred_val = predict(X_val_scaled, W1, b1, W2, b2)
            val_accuracy = accuracy(y_val, y_pred_val)
            grid_accuracy[i, j, k] = val_accuracy
            if val_accuracy > best_accuracy:
                best_accuracy = val_accuracy
                best_lr = lr
                best_hidden_size = hidden_size
                best_num_iterations = num_iterations
            plt.figure(figsize=(6, 4))
            plt.bar(['Validation Accuracy'], [val_accuracy])
            plt.ylim(0, 1)
            plt.title(f"LR: {lr}, Hidden Size: {hidden_size}, Iterations:␣
 ↪{num_iterations}\nAccuracy: {val_accuracy:.3f}")
            plt.show()

W1, b1, W2, b2 = train(X_train_val_scaled, y_train_val, intake_bounus_q,␣
 ↪best_hidden_size, output_size, best_lr, best_num_iterations)
y_pred_test = predict(X_test_scaled, W1, b1, W2, b2)
test_accuracy = accuracy(y_test, y_pred_test)
print(f"Best Hyperparameters: LR = {best_lr}, Hidden Size = {best_hidden_size},␣
 ↪Iterations = {best_num_iterations}")
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

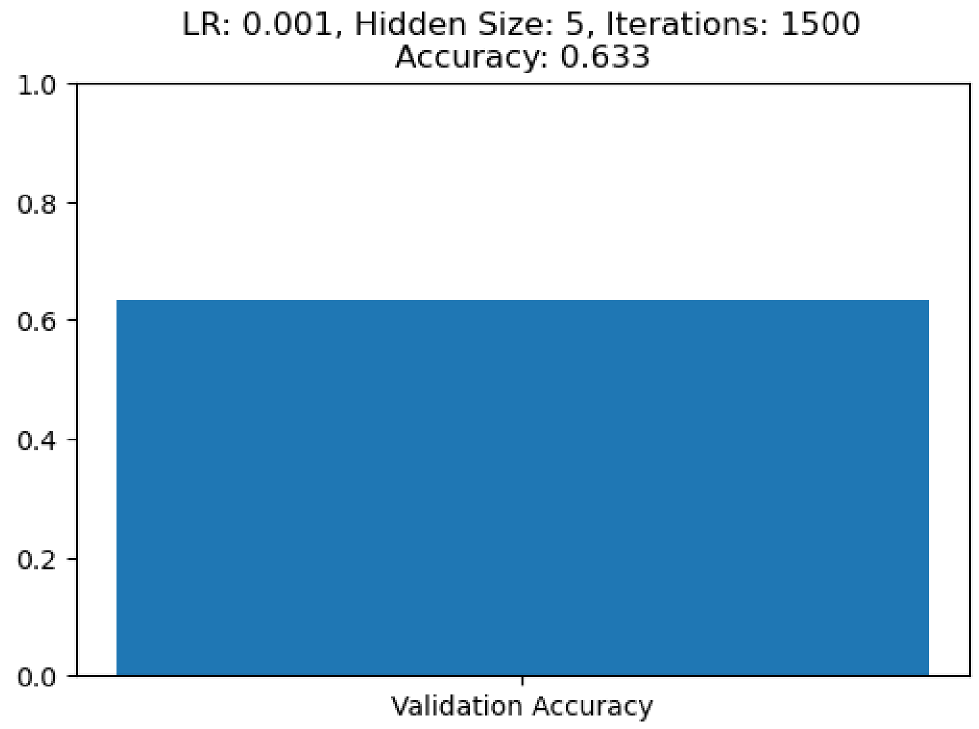
```
Iteration 0 loss: 0.7077810504574003
Iteration 100 loss: 0.7049004671295928
Iteration 200 loss: 0.7025437183793766
Iteration 300 loss: 0.7005755382465012
Iteration 400 loss: 0.6989679654321225
```



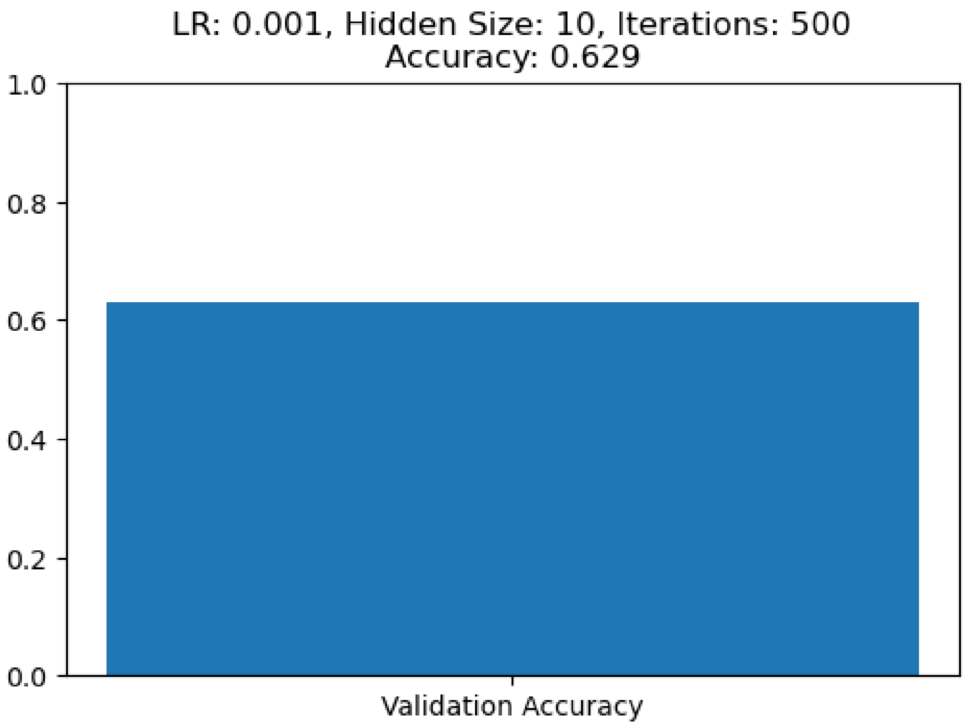
```
Iteration 0 loss: 0.673186605971687
Iteration 100 loss: 0.670677995695503
Iteration 200 loss: 0.6685140049148193
Iteration 300 loss: 0.6666371282719128
Iteration 400 loss: 0.664987505374934
Iteration 500 loss: 0.6635029977876025
Iteration 600 loss: 0.6621454632782949
Iteration 700 loss: 0.6609075112950792
Iteration 800 loss: 0.6597425393720434
Iteration 900 loss: 0.6586242341062661
```



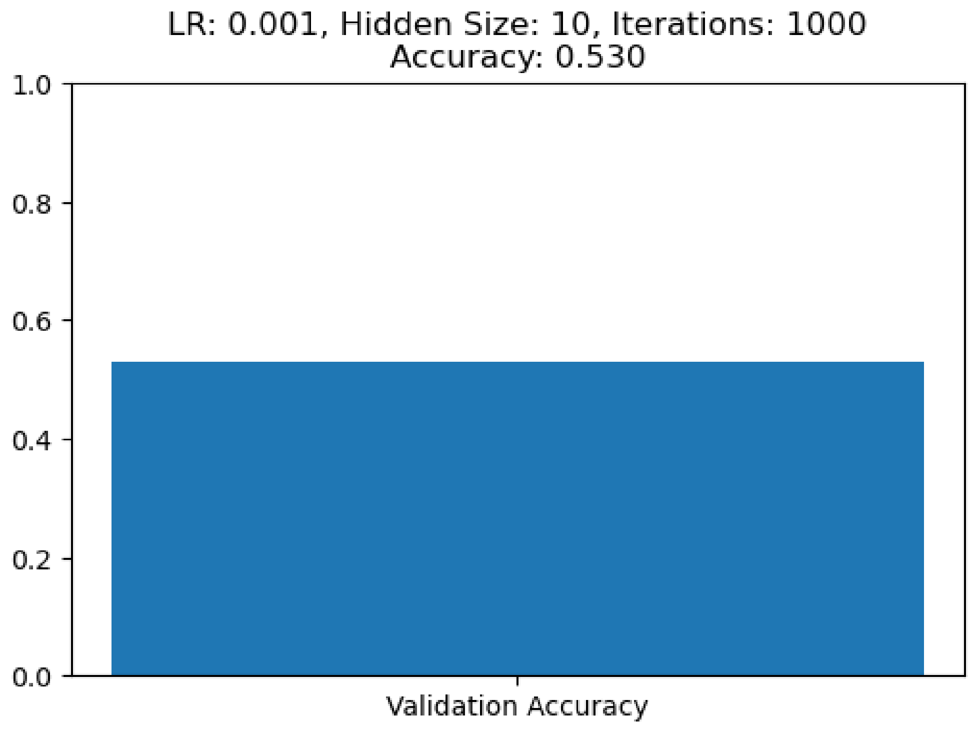
```
Iteration 0 loss: 0.6996375087882329
Iteration 100 loss: 0.6920081770451947
Iteration 200 loss: 0.685998605120091
Iteration 300 loss: 0.6811555391076068
Iteration 400 loss: 0.6771800824903116
Iteration 500 loss: 0.674002654887235
Iteration 600 loss: 0.6712768028085475
Iteration 700 loss: 0.6689214731316154
Iteration 800 loss: 0.666916035623153
Iteration 900 loss: 0.6651734394057938
Iteration 1000 loss: 0.6636604824633606
Iteration 1100 loss: 0.6622163862342274
Iteration 1200 loss: 0.6608115356965256
Iteration 1300 loss: 0.6594432448153439
Iteration 1400 loss: 0.6581121197794211
```



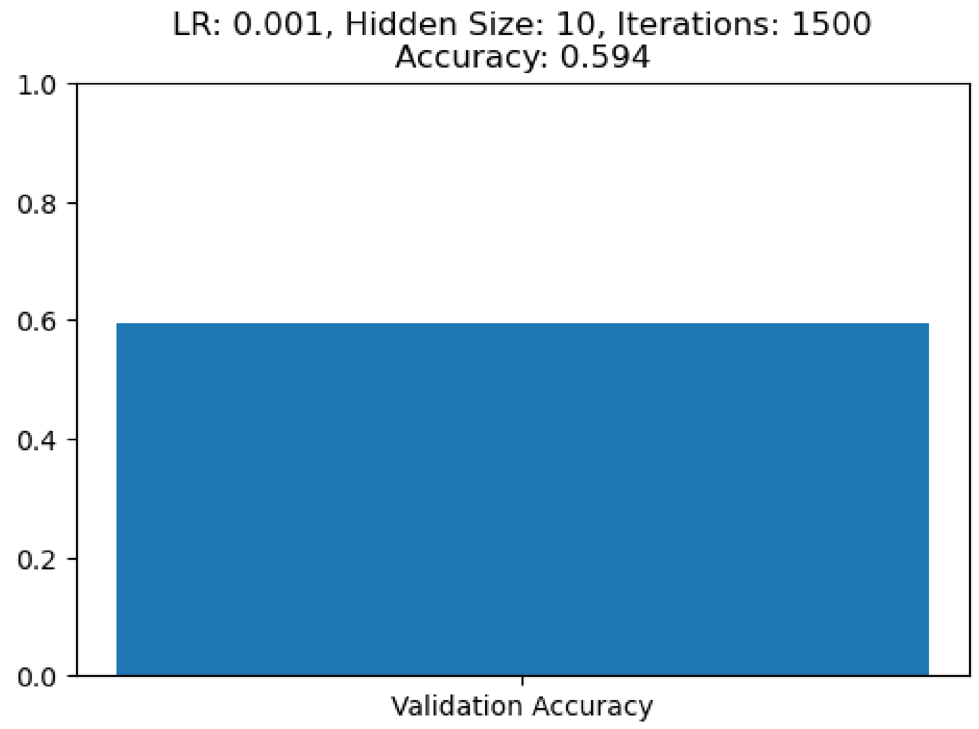
```
Iteration 0 loss: 0.811963049644403
Iteration 100 loss: 0.6922081042049314
Iteration 200 loss: 0.6544632646776471
Iteration 300 loss: 0.6417367473974172
Iteration 400 loss: 0.6363921116715865
```



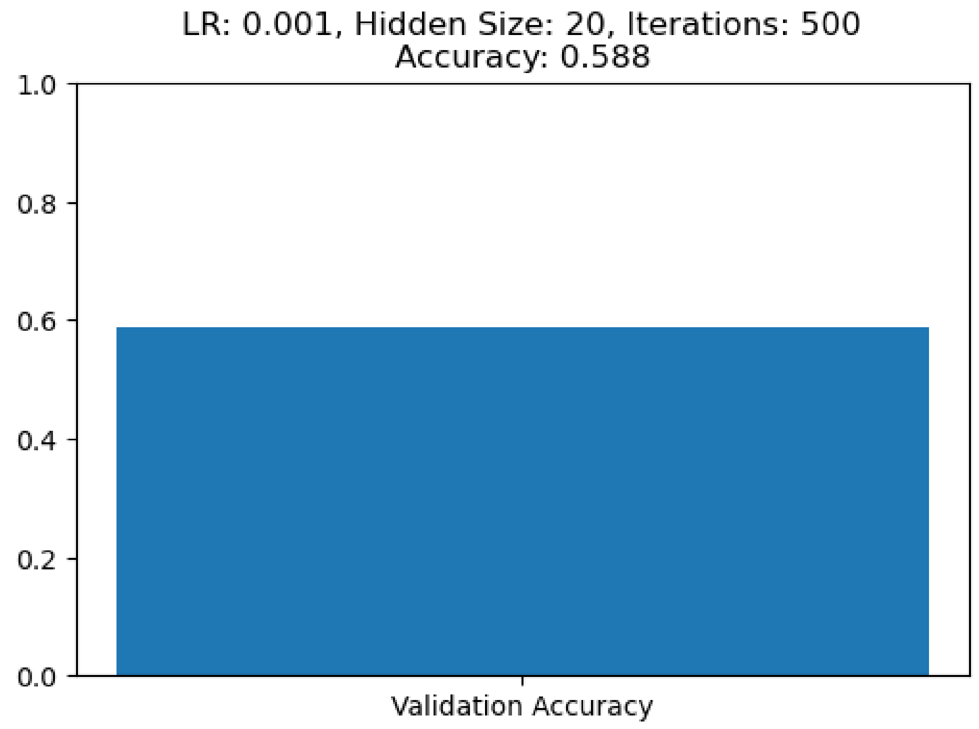
```
Iteration 0 loss: 0.8213478816436539
Iteration 100 loss: 0.7752275072649354
Iteration 200 loss: 0.7481176634947256
Iteration 300 loss: 0.7315028987548081
Iteration 400 loss: 0.7204161156951177
Iteration 500 loss: 0.7127978979068628
Iteration 600 loss: 0.7071431765016563
Iteration 700 loss: 0.702844107152176
Iteration 800 loss: 0.6993209725494435
Iteration 900 loss: 0.6963326379686307
```



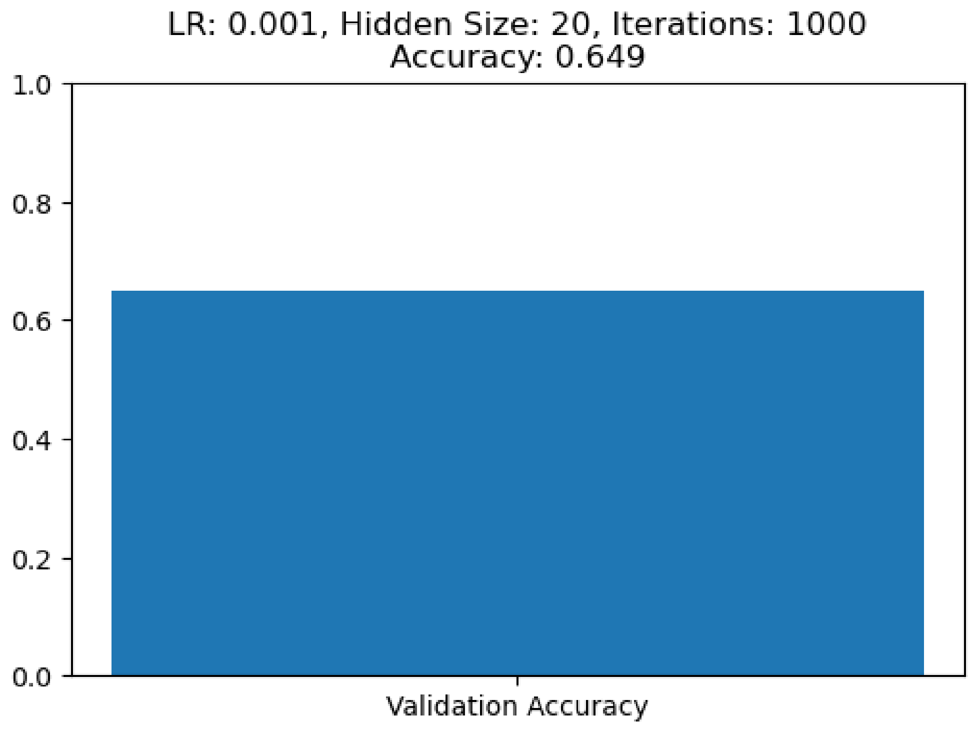
```
Iteration 0 loss: 0.7536351237862905
Iteration 100 loss: 0.7172154417983991
Iteration 200 loss: 0.6959300216494521
Iteration 300 loss: 0.6825131246088834
Iteration 400 loss: 0.6735936918795116
Iteration 500 loss: 0.667424363497826
Iteration 600 loss: 0.6628619188797247
Iteration 700 loss: 0.6591458939491733
Iteration 800 loss: 0.6559089957316905
Iteration 900 loss: 0.6530978450403331
Iteration 1000 loss: 0.6506234102094175
Iteration 1100 loss: 0.648260589636231
Iteration 1200 loss: 0.6461991773308154
Iteration 1300 loss: 0.6444058787397342
Iteration 1400 loss: 0.6426665314677477
```



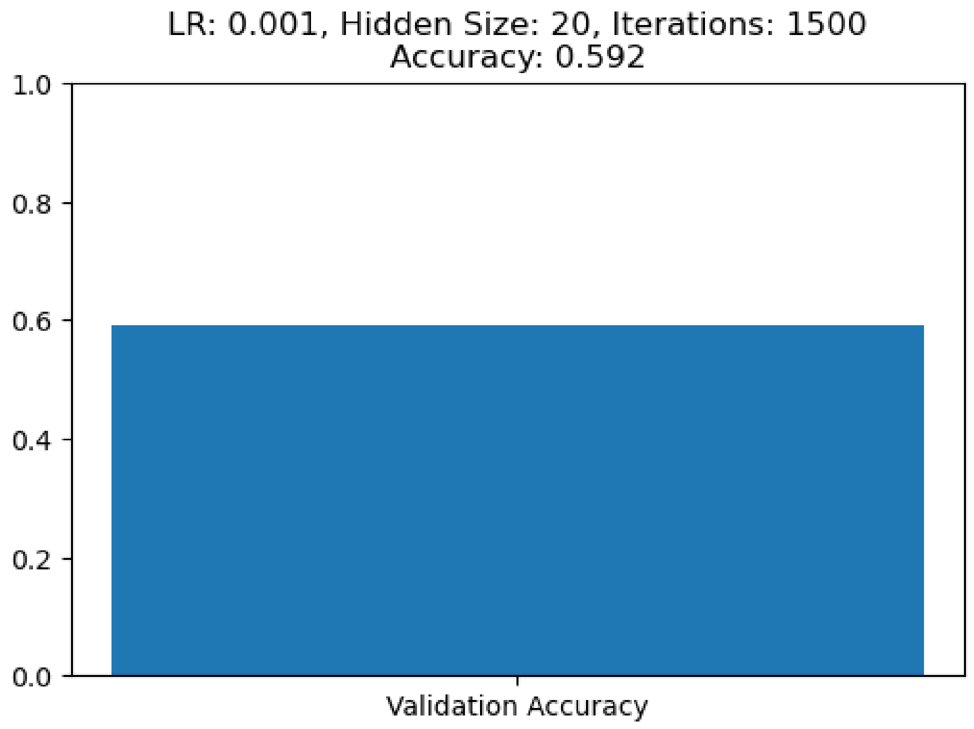
```
Iteration 0 loss: 0.698019402493473
Iteration 100 loss: 0.6856346631186261
Iteration 200 loss: 0.6794433812442011
Iteration 300 loss: 0.674513683534337
Iteration 400 loss: 0.6699776165794301
```



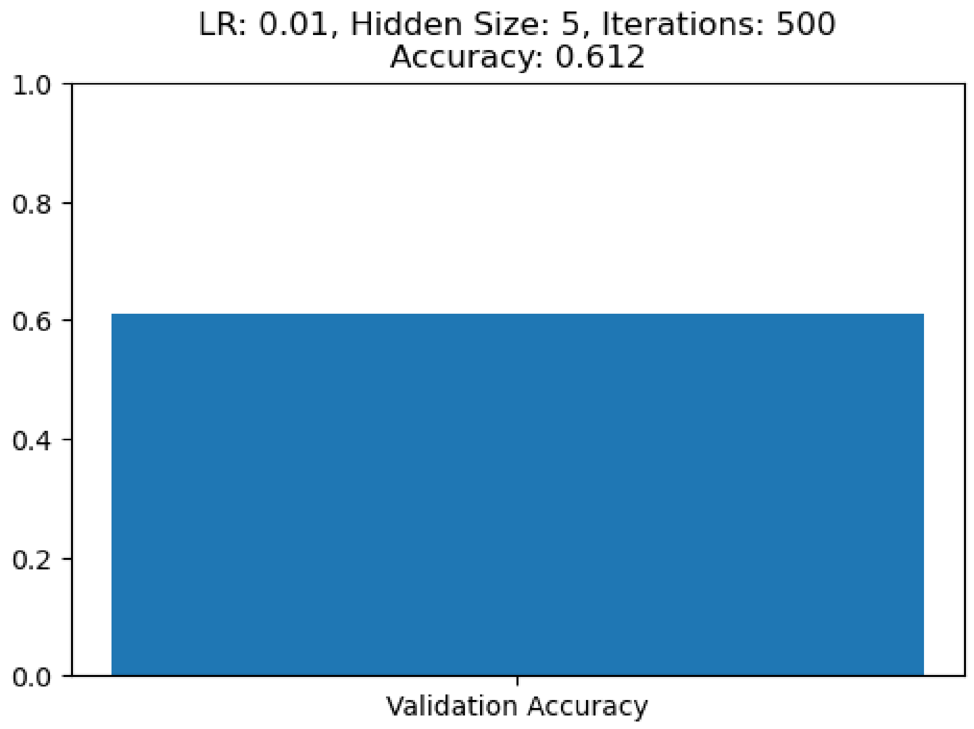
```
Iteration 0 loss: 0.7146504923002666
Iteration 100 loss: 0.677159787864748
Iteration 200 loss: 0.6642951516917677
Iteration 300 loss: 0.6587688820800931
Iteration 400 loss: 0.655383806093171
Iteration 500 loss: 0.6526449109417497
Iteration 600 loss: 0.6501066041249681
Iteration 700 loss: 0.6476649084382633
Iteration 800 loss: 0.6452876207777603
Iteration 900 loss: 0.6429623592055606
```



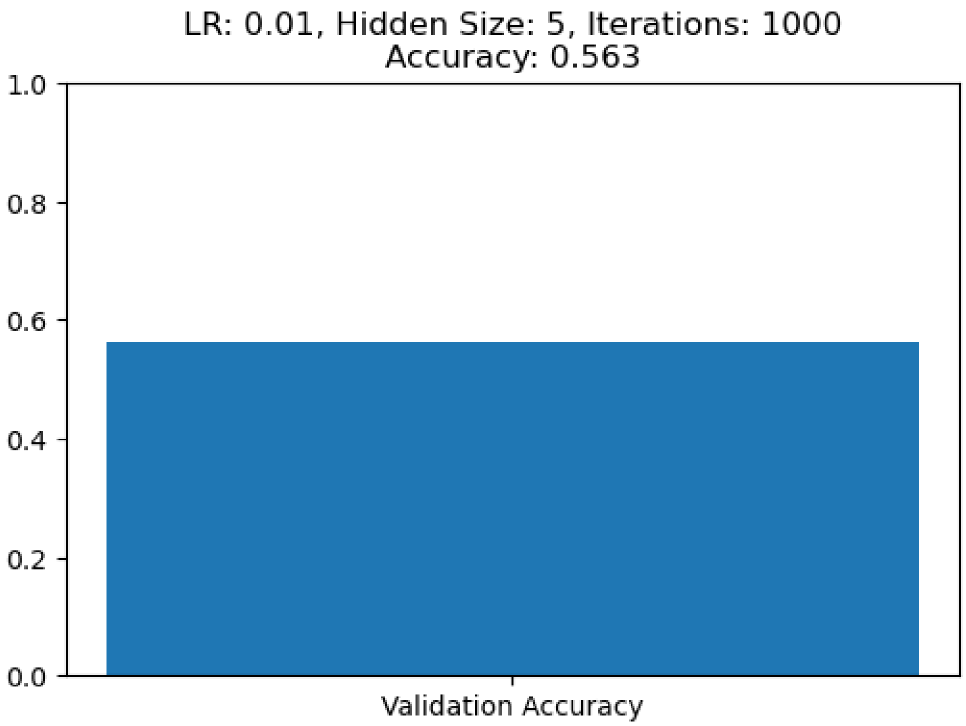
```
Iteration 0 loss: 0.731665034499192
Iteration 100 loss: 0.7097091962238142
Iteration 200 loss: 0.6969429020829407
Iteration 300 loss: 0.6890101975031658
Iteration 400 loss: 0.6835818408311236
Iteration 500 loss: 0.6795847651546505
Iteration 600 loss: 0.6764078509483947
Iteration 700 loss: 0.673674798927696
Iteration 800 loss: 0.6712578721367796
Iteration 900 loss: 0.6691114909223305
Iteration 1000 loss: 0.667130088034501
Iteration 1100 loss: 0.6652495506818272
Iteration 1200 loss: 0.6634493292971699
Iteration 1300 loss: 0.6617296618772344
Iteration 1400 loss: 0.6600641848617268
```



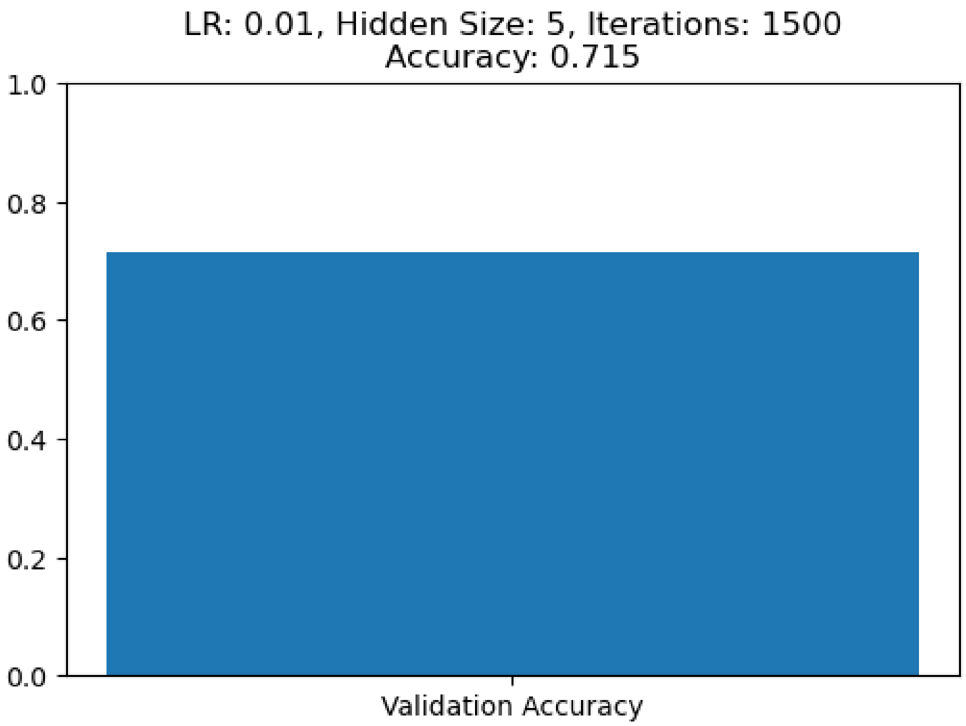
```
Iteration 0 loss: 0.6997280243398172
Iteration 100 loss: 0.6798665629711603
Iteration 200 loss: 0.6725128831822874
Iteration 300 loss: 0.6659105263883796
Iteration 400 loss: 0.6584213735138648
```



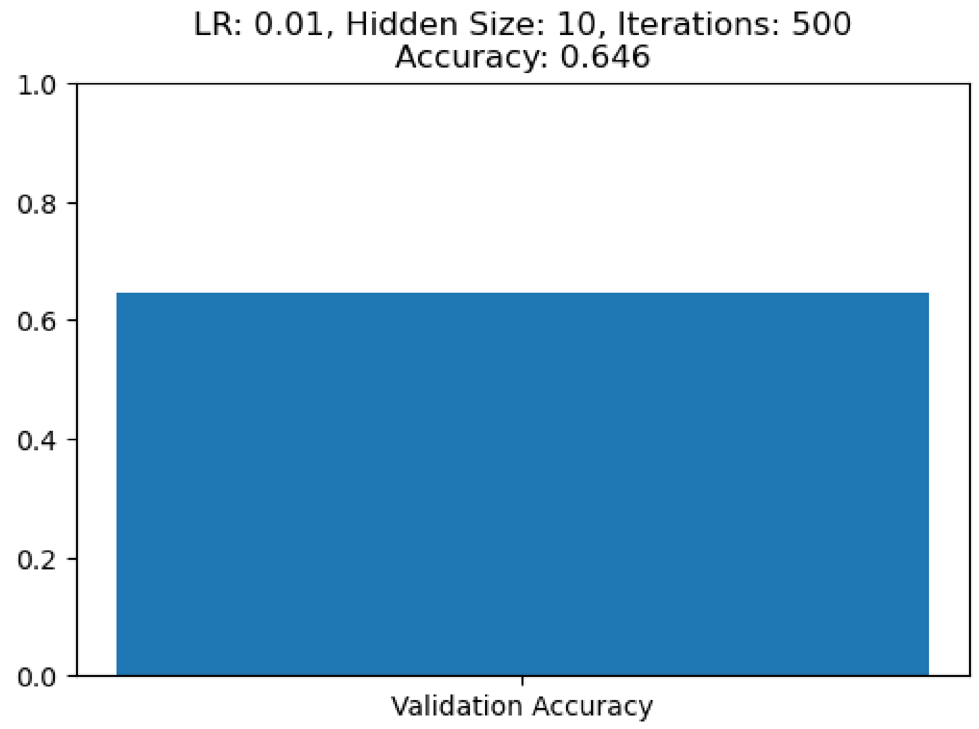
```
Iteration 0 loss: 0.8340931372975258
Iteration 100 loss: 0.6987106419607346
Iteration 200 loss: 0.6865151145948486
Iteration 300 loss: 0.6794712559859991
Iteration 400 loss: 0.6736061038838901
Iteration 500 loss: 0.6680390221892414
Iteration 600 loss: 0.6620009036143829
Iteration 700 loss: 0.6555771167235435
Iteration 800 loss: 0.6483985620643148
Iteration 900 loss: 0.6404748659638877
```



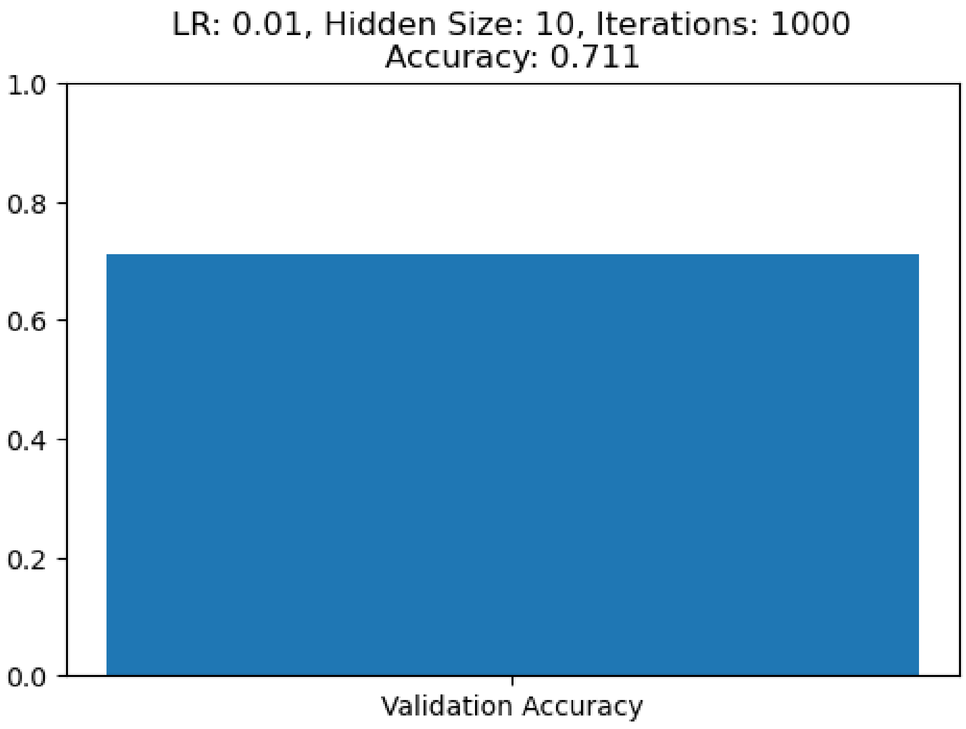
```
Iteration 0 loss: 0.9496450886085348
Iteration 100 loss: 0.6844162281082069
Iteration 200 loss: 0.6642600758668707
Iteration 300 loss: 0.6482582961079653
Iteration 400 loss: 0.6293340350152463
Iteration 500 loss: 0.6082187938228375
Iteration 600 loss: 0.5870607042199293
Iteration 700 loss: 0.5696819940971775
Iteration 800 loss: 0.5557295301183924
Iteration 900 loss: 0.5446385411951339
Iteration 1000 loss: 0.5358782051047251
Iteration 1100 loss: 0.5288458376101911
Iteration 1200 loss: 0.5231570630315832
Iteration 1300 loss: 0.5184825917019297
Iteration 1400 loss: 0.5145727070475237
```



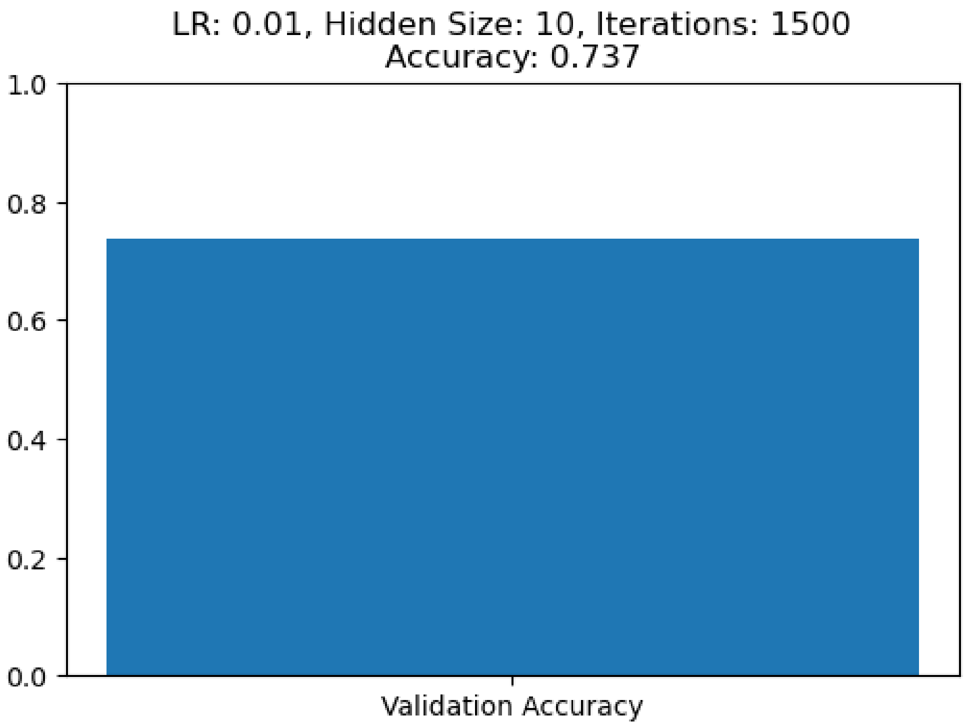
```
Iteration 0 loss: 0.8192494086364298
Iteration 100 loss: 0.6876864575189932
Iteration 200 loss: 0.6639302951137839
Iteration 300 loss: 0.6439072690566939
Iteration 400 loss: 0.6254443141912153
```



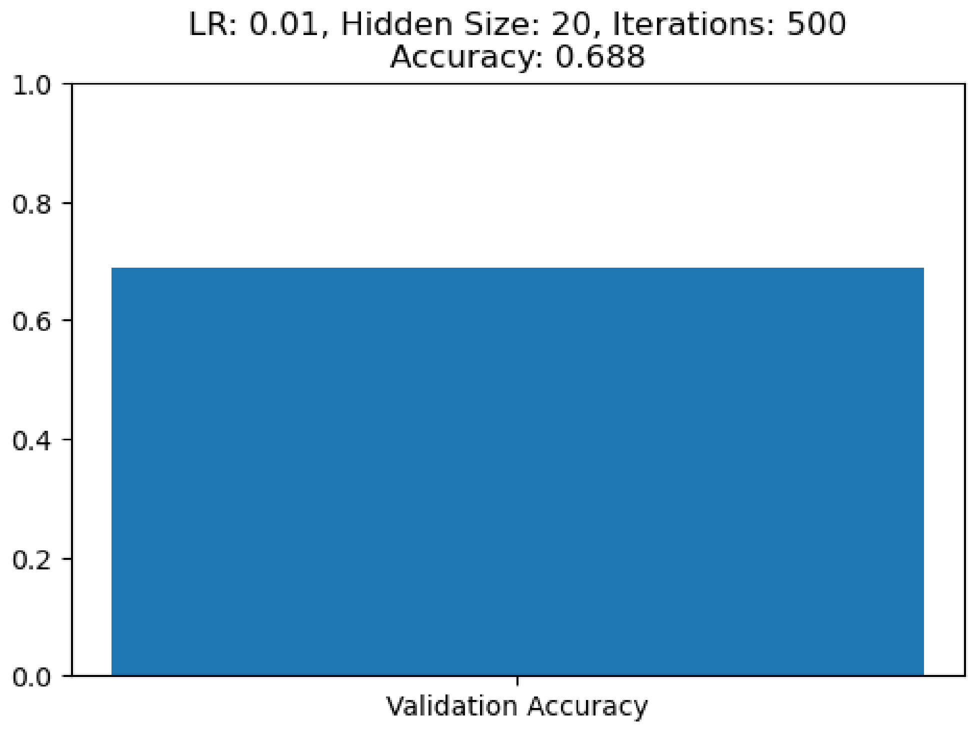
```
Iteration 0 loss: 0.6930219497995634
Iteration 100 loss: 0.6505047989830267
Iteration 200 loss: 0.6323788863659385
Iteration 300 loss: 0.6164550561399147
Iteration 400 loss: 0.601991024710912
Iteration 500 loss: 0.5885438743787181
Iteration 600 loss: 0.57609342430214
Iteration 700 loss: 0.5648545611966744
Iteration 800 loss: 0.5549419380621786
Iteration 900 loss: 0.5463823390437307
```



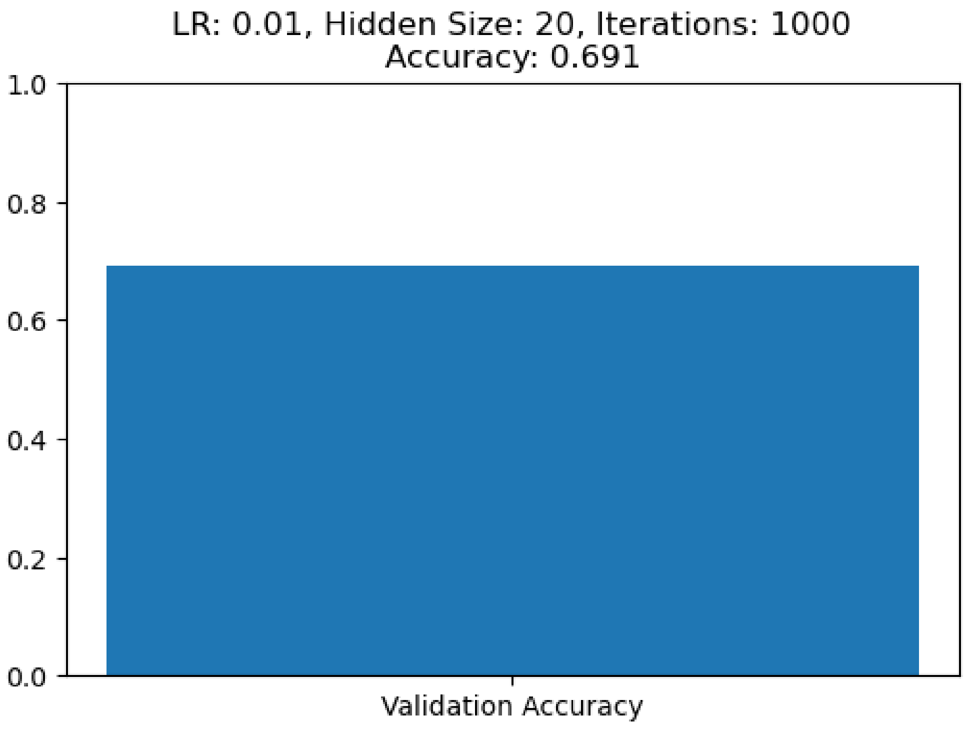
```
Iteration 0 loss: 0.9851341904610532
Iteration 100 loss: 0.7032071698953454
Iteration 200 loss: 0.6530320157216243
Iteration 300 loss: 0.6206698950669057
Iteration 400 loss: 0.5972192472821283
Iteration 500 loss: 0.5789392680116712
Iteration 600 loss: 0.5642566641674547
Iteration 700 loss: 0.5524431884785401
Iteration 800 loss: 0.5427556408965284
Iteration 900 loss: 0.5348752687416296
Iteration 1000 loss: 0.5283869400248746
Iteration 1100 loss: 0.5229070485801967
Iteration 1200 loss: 0.5182263081460003
Iteration 1300 loss: 0.5142130252542672
Iteration 1400 loss: 0.510782723637786
```



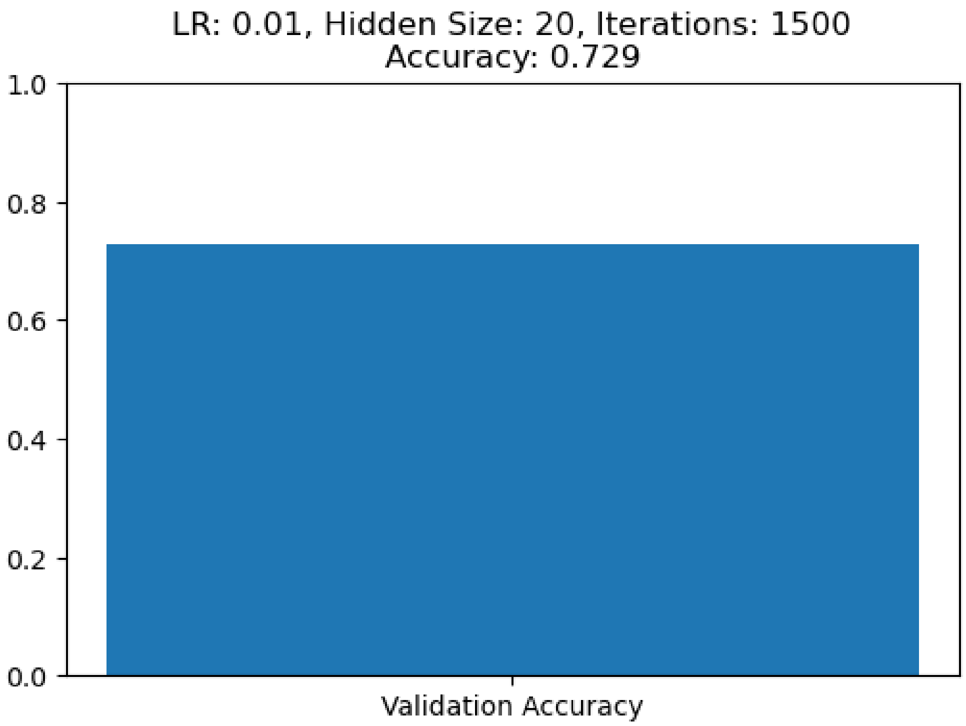
```
Iteration 0 loss: 0.8227535709503347
Iteration 100 loss: 0.6563383433637244
Iteration 200 loss: 0.6313219548012208
Iteration 300 loss: 0.6103750554281883
Iteration 400 loss: 0.5926653967857562
```



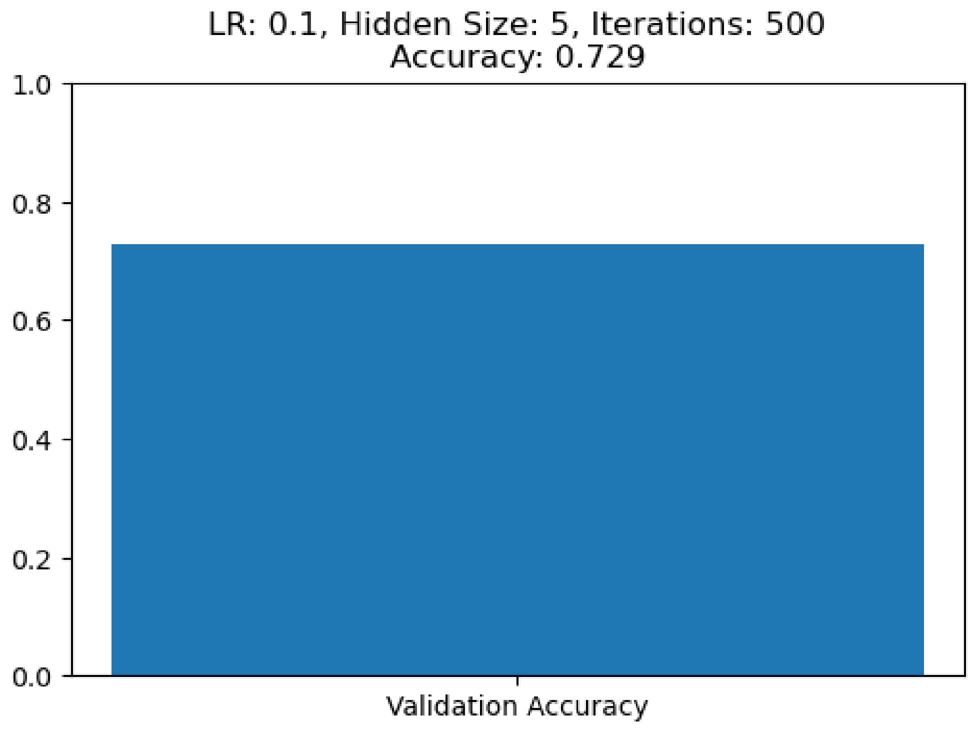
```
Iteration 0 loss: 0.7351014829676273
Iteration 100 loss: 0.6905789195575961
Iteration 200 loss: 0.6600503414362867
Iteration 300 loss: 0.6368849173423187
Iteration 400 loss: 0.6179066951440011
Iteration 500 loss: 0.6016552060650694
Iteration 600 loss: 0.5873272793214173
Iteration 700 loss: 0.574713558137866
Iteration 800 loss: 0.5636443589181033
Iteration 900 loss: 0.5541235321183118
```



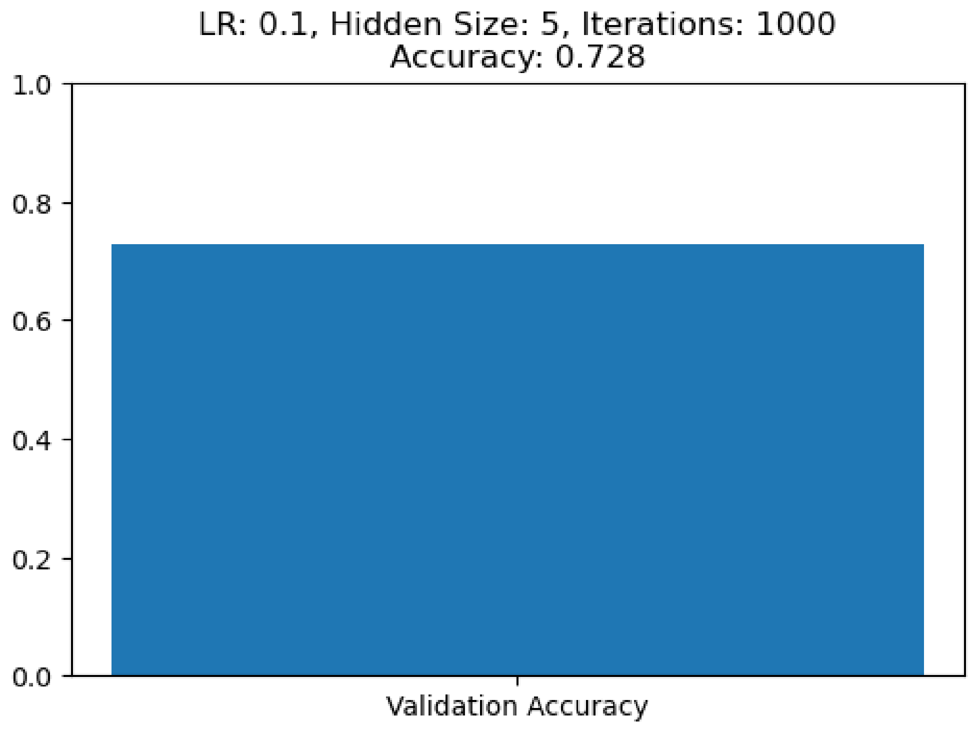
```
Iteration 0 loss: 0.7217740120157314
Iteration 100 loss: 0.6548815415243432
Iteration 200 loss: 0.6262081662551765
Iteration 300 loss: 0.6057909305422498
Iteration 400 loss: 0.5893006919389663
Iteration 500 loss: 0.5753162888760179
Iteration 600 loss: 0.5634912538586
Iteration 700 loss: 0.553532414569359
Iteration 800 loss: 0.5451778492254764
Iteration 900 loss: 0.5381928267920179
Iteration 1000 loss: 0.5323070901718953
Iteration 1100 loss: 0.5273207896644552
Iteration 1200 loss: 0.5230428708576118
Iteration 1300 loss: 0.5193514200835849
Iteration 1400 loss: 0.5161601144496142
```



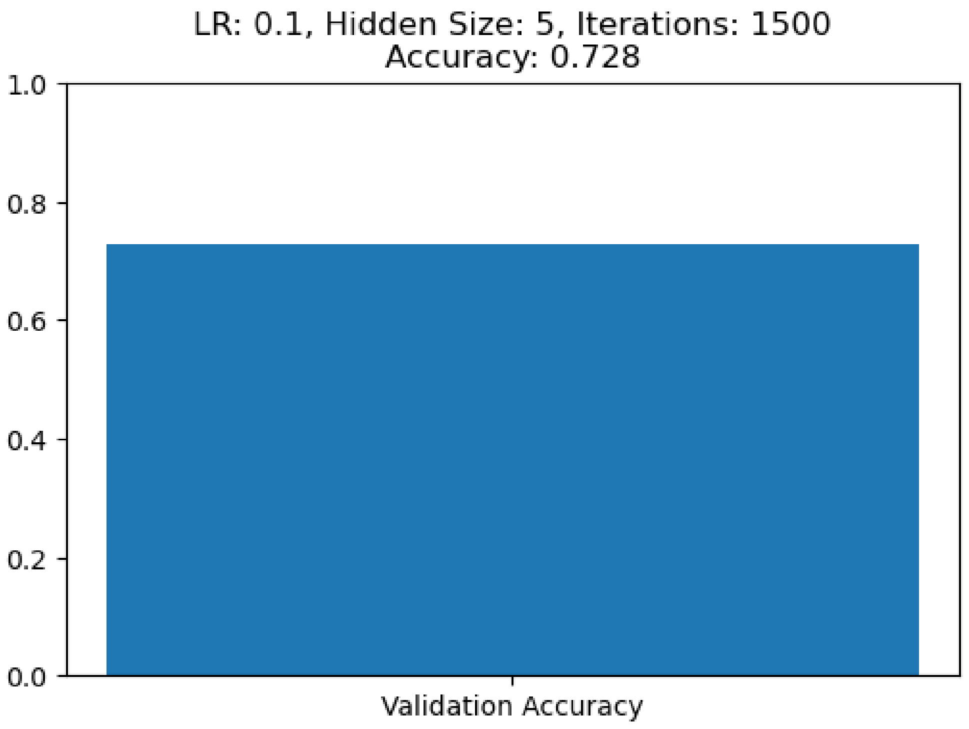
```
Iteration 0 loss: 1.2631560516910934
Iteration 100 loss: 0.6789331637575471
Iteration 200 loss: 0.5886599622591149
Iteration 300 loss: 0.5146419591222895
Iteration 400 loss: 0.4968040349450542
```



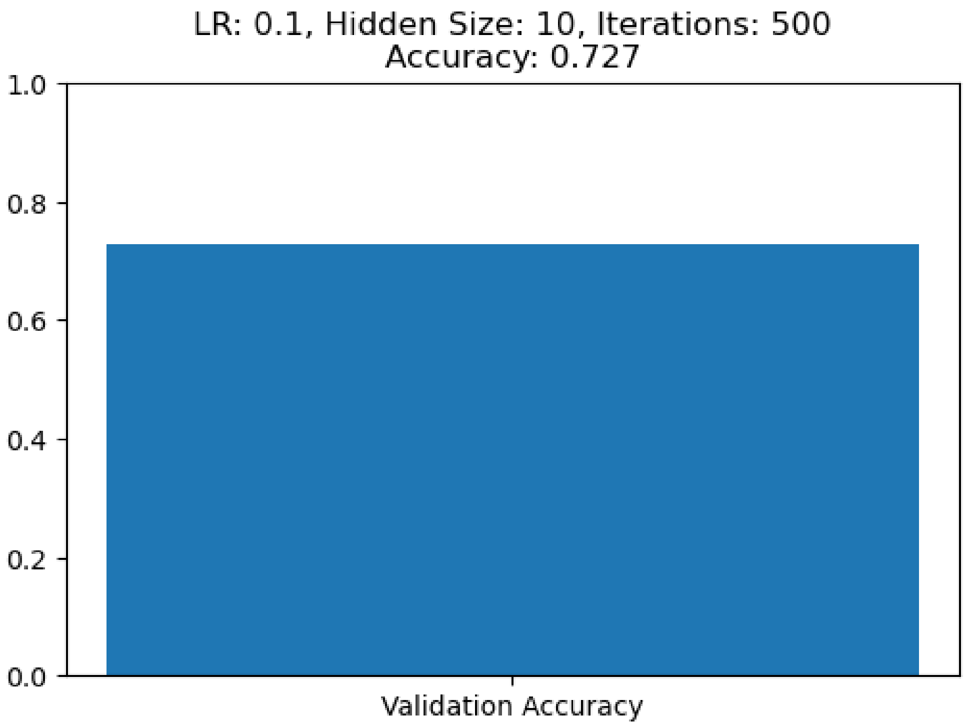
```
Iteration 0 loss: 0.7728379587261239
Iteration 100 loss: 0.5353049942992427
Iteration 200 loss: 0.5058169588550296
Iteration 300 loss: 0.49663102555619854
Iteration 400 loss: 0.49184541620015537
Iteration 500 loss: 0.4881440674519592
Iteration 600 loss: 0.48567211324015935
Iteration 700 loss: 0.48342337574773914
Iteration 800 loss: 0.48156378826059537
Iteration 900 loss: 0.4799054699702672
```



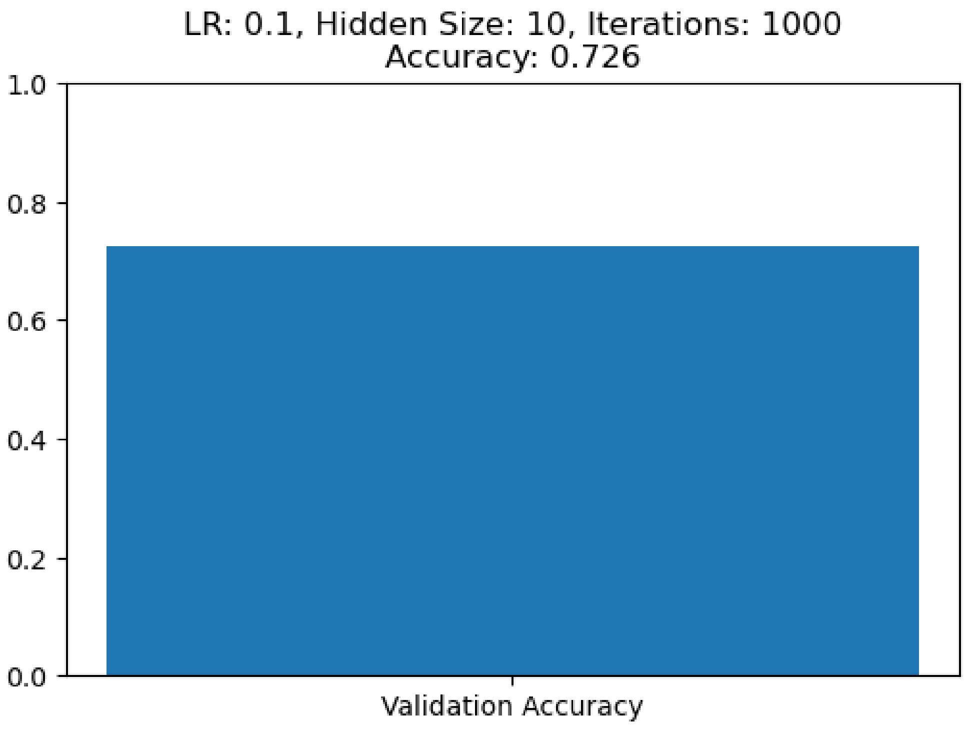
```
Iteration 0 loss: 1.0803270892346457
Iteration 100 loss: 0.663378558427417
Iteration 200 loss: 0.5566602809204547
Iteration 300 loss: 0.5099010264556545
Iteration 400 loss: 0.49629935667764413
Iteration 500 loss: 0.48929165955170495
Iteration 600 loss: 0.48498718785949824
Iteration 700 loss: 0.48204921295155173
Iteration 800 loss: 0.479806917930977
Iteration 900 loss: 0.4779909839034091
Iteration 1000 loss: 0.47662727388243253
Iteration 1100 loss: 0.47550193085695136
Iteration 1200 loss: 0.4742961528528447
Iteration 1300 loss: 0.47305381015383813
Iteration 1400 loss: 0.47157165524506844
```



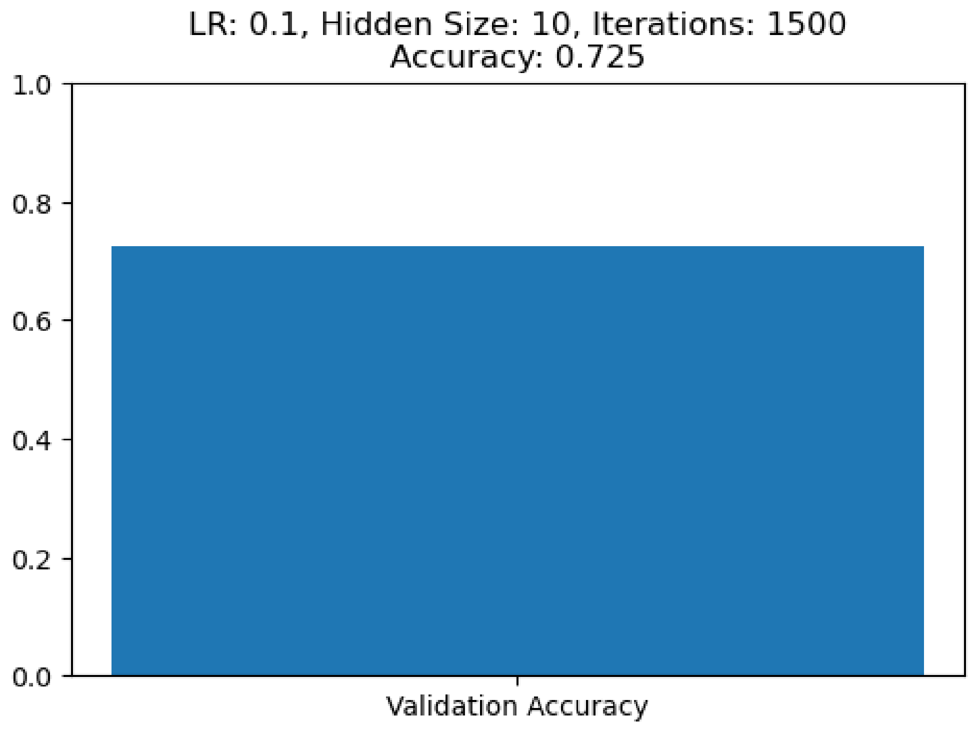
```
Iteration 0 loss: 0.7543639014187175
Iteration 100 loss: 0.5706171106664504
Iteration 200 loss: 0.5149637131806387
Iteration 300 loss: 0.4997153499034939
Iteration 400 loss: 0.49204916616166394
```



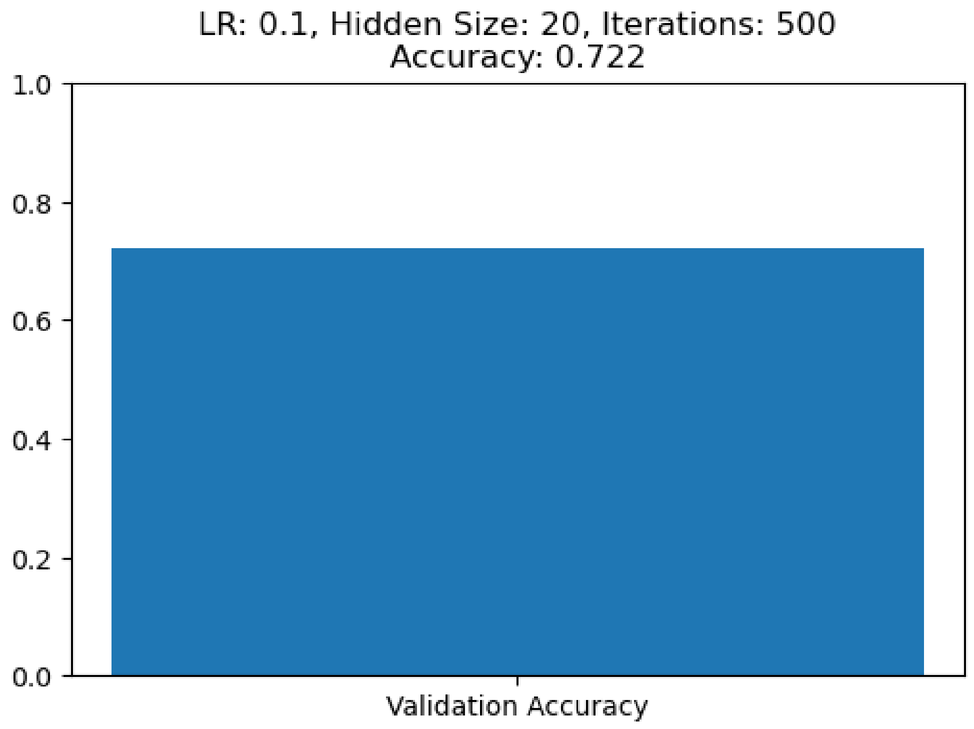
```
Iteration 0 loss: 0.7020332548390127
Iteration 100 loss: 0.5577247764087288
Iteration 200 loss: 0.5080406048628222
Iteration 300 loss: 0.49194192377488827
Iteration 400 loss: 0.48338461843229136
Iteration 500 loss: 0.47775305769811516
Iteration 600 loss: 0.47344005811807227
Iteration 700 loss: 0.46999786666929255
Iteration 800 loss: 0.46690306591671454
Iteration 900 loss: 0.4640400747232867
```



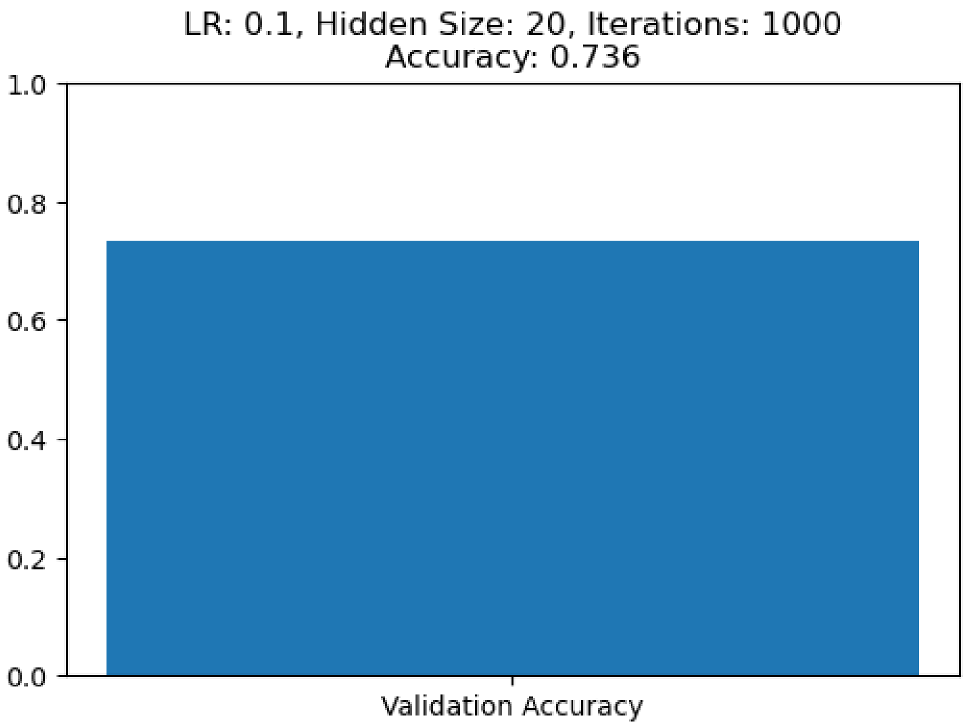
```
Iteration 0 loss: 0.7391874021450191
Iteration 100 loss: 0.5543990397494422
Iteration 200 loss: 0.5156987657801881
Iteration 300 loss: 0.4997668697421662
Iteration 400 loss: 0.4915579159575371
Iteration 500 loss: 0.4864191243410894
Iteration 600 loss: 0.4824333433413218
Iteration 700 loss: 0.4792694855432706
Iteration 800 loss: 0.47649263589231067
Iteration 900 loss: 0.47356342805627505
Iteration 1000 loss: 0.47063130906739015
Iteration 1100 loss: 0.46795593025431265
Iteration 1200 loss: 0.46532996880612376
Iteration 1300 loss: 0.46266874617958953
Iteration 1400 loss: 0.459866715353828
```



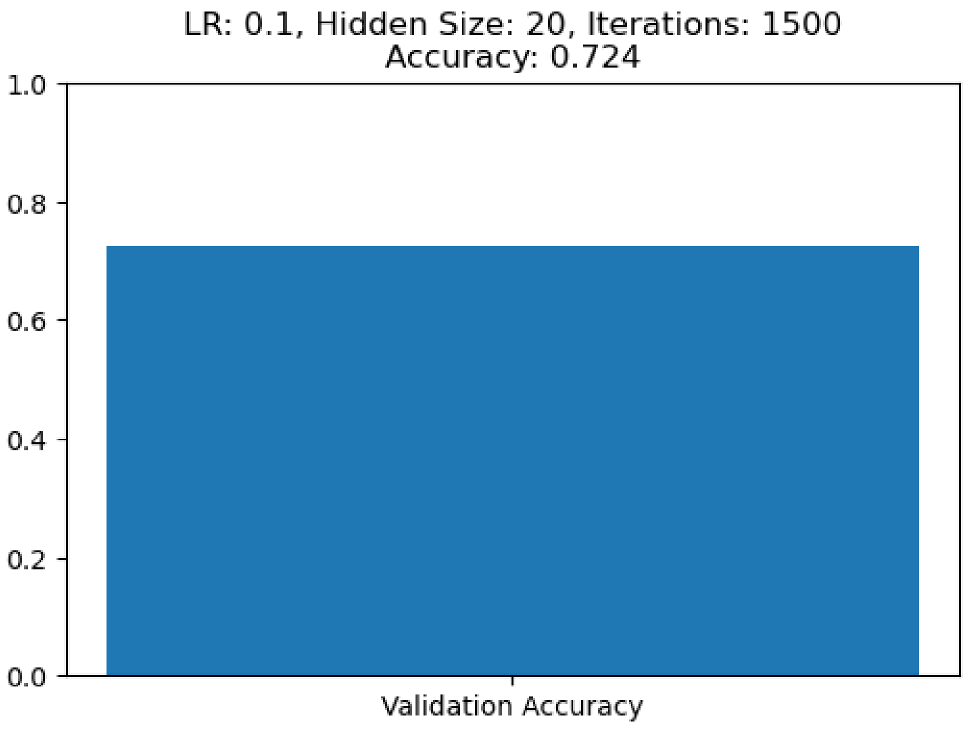
```
Iteration 0 loss: 0.7949882910983538
Iteration 100 loss: 0.5385208191182943
Iteration 200 loss: 0.500979540557517
Iteration 300 loss: 0.4891243802992886
Iteration 400 loss: 0.48263215792140585
```



```
Iteration 0 loss: 0.8323934181307866
Iteration 100 loss: 0.5767448036682207
Iteration 200 loss: 0.5158844140145813
Iteration 300 loss: 0.49739473662339445
Iteration 400 loss: 0.48787835184429656
Iteration 500 loss: 0.481578204437845
Iteration 600 loss: 0.4765029609444549
Iteration 700 loss: 0.4716019544205219
Iteration 800 loss: 0.4669459946565886
Iteration 900 loss: 0.46269816530554236
```



```
Iteration 0 loss: 0.734356968571778
Iteration 100 loss: 0.5606362772037324
Iteration 200 loss: 0.5095492575328204
Iteration 300 loss: 0.4918507523902566
Iteration 400 loss: 0.4822122841150319
Iteration 500 loss: 0.47516545246564595
Iteration 600 loss: 0.47011394806875934
Iteration 700 loss: 0.46557496957671857
Iteration 800 loss: 0.4610929348826874
Iteration 900 loss: 0.45685499307975264
Iteration 1000 loss: 0.45268288158582537
Iteration 1100 loss: 0.4483721975566234
Iteration 1200 loss: 0.44399160774654167
Iteration 1300 loss: 0.43958270225076457
Iteration 1400 loss: 0.43526569723260156
```



```
Iteration 0 loss: 0.703948759876753
Iteration 100 loss: 0.6508987733497837
Iteration 200 loss: 0.6204332271095593
Iteration 300 loss: 0.5961842702094589
Iteration 400 loss: 0.5776717171813968
Iteration 500 loss: 0.5634378591155507
Iteration 600 loss: 0.552610789089813
Iteration 700 loss: 0.544232597937282
Iteration 800 loss: 0.5376634236425681
Iteration 900 loss: 0.532420714579017
Iteration 1000 loss: 0.5280941628185275
Iteration 1100 loss: 0.5244987199100777
Iteration 1200 loss: 0.521476377088106
Iteration 1300 loss: 0.5188870368855338
Iteration 1400 loss: 0.5166794019316044
Best Hyperparameters: LR = 0.01, Hidden Size = 10, Iterations = 1500
Test Accuracy: 66.00%
```

Hence we can see the best combination is from hyperparamters: LR = 0.01, Hidden Size = 10, Iterations = 1500

1) Data Preprocessing:

The dataset was split into training and test sets. The training set was further divided into training and validation subsets. Data scaling was performed using the mean and standard deviation of the training subset to standardize the training, validation, and test datasets.

2) Hyperparameter Tuning: A grid search approach was employed to iterate over various combinations of hyperparameters. The hyperparameters considered were: Learning Rate (LR) Hidden Layer Size Number of Iterations The chosen values for exploration were: Learning Rates: [0.001, 0.01, 0.1] Hidden Sizes: [5, 10, 20] Number of Iterations: [500, 1000, 1500]

3) Model Training and Validation: For each combination of hyperparameters, the neural network model was trained using the training subset. The model's performance was then evaluated on the validation subset. This process helped in identifying the best hyperparameter set based on validation accuracy.

4) Model Selection: The best-performing model was identified with the following hyperparameters: Learning Rate: 0.01 Hidden Layer Size: 10 Number of Iterations: 1500 This combination provided the highest accuracy on the validation subset.

5) Testing and Evaluation: The final model was evaluated on the test set. Test Accuracy: The model achieved an accuracy of 66.00% on the test set.