Career Recommendation System

Choose Your Ideal Career

HITEN PATIL

1. Project Overview

1.1 Objective

The Career Recommendation System is a Flask-based web application designed to recommend personalized career paths based on user inputs: technical skills (skills), professional interests (interests), and experience level (experience). Leveraging machine learning models trained on **career_data_new.csv**, the system predicts careers such as **"Data Scientist"** or **"Web Developer."** It supports English and Gujarati interfaces, with the frontend managing language switching via data-gujarati attributes. The project aims to guide students and professionals in career planning.

1.2 Scope

- **Backend:** Flask REST API using **custom Decision Tree** and **K-Nearest Neighbors (KNN)** models.
- **Frontend:** HTML form **(index.html)** with CSS **(styles.css)** and JavaScript **(script.js)** for input collection and result display.
- **Dataset**: **career_data_new.csv**, containing **Skill, Interests, Experience, and Career columns.**
- **Features:**
    - Predicts careers based on user inputs.
    - Combines predictions from **KNN** and **Decision Tree** into a consensus result.
    - Logs debugging information to app.log.
- **Deliverables:**
    - Functional Flask application **(app.py)**.
    - Trained models **(decision_tree_model.pkl, knn_model.pkl).**
    - User-friendly frontend with bilingual support.
    - Documentation of model training, selection, and issue resolution.

1.3 Development Timeline

Phase 1: Trained five machine learning models on **career_data_new.csv.**

Phase 2: Evaluated models and selected **KNN** and **Decision Tree** based on **accuracy.**

Phase 3: Developed Flask backend and frontend.

2. Methodology

2.1 Dataset

The dataset, career_data_new.csv, served as the basis for training and inference:

- **Columns:**
  - **Skill:** Technical skills (e.g., Python, Java).
  - **Interests:** Professional interests (e.g., AI Research, Web Development).
  - **Experience:** Experience level (Beginner, Intermediate, Advanced).
  - **Career:** Target career path (e.g., Data Scientist, Web Developer).
- **Role:**
  - **Training:** Provided features and labels for model training.
  - **Inference:** Generated category mappings for preprocessing user inputs.

2.2 Model Training

Five machine learning models were trained to predict careers based on encoded inputs (Skill, Interests, Experience):

1. **Decision Tree:**
   - Custom implementation (DecisionTreeScratch) using Gini impurity.
   - Parameters: max_depth=5, min_samples_split=2.
   - Advantages: Interpretable, effective for categorical data.
2. **K-Nearest Neighbors (KNN):**
   - Custom implementation (KNNScratch) using Euclidean distance.
   - Parameters: k=5.
   - Advantages: Simple, high performance on small datasets.
3. **Logistic Regression:**
   - Advantages: Fast, suitable for linear relationships.
4. **Naive Bayes:**
   - Advantages: Probabilistic, efficient for categorical data.
5. **Random Forest:**
   - Advantages: Robust, reduces overfitting.

Training Process:

- **Preprocessing:**
    - **Encoded categorical variables:**
        - Skill and Interests to numerical codes using pandas.astype('category').cat.categories.
        - Experience to Beginner=0, Intermediate=1, Advanced=2.
    - Split data into 80% training and 20% testing sets using train_test_split.
- **Training:** Each model was trained on the encoded training set.
- **Evaluation:** Models were evaluated on the test set using accuracy.
- **Storage:** Models saved as .pkl files for the selected models.

TrainProject.py

```
import pandas as pd
import numpy as np
import pickle
from collections import Counter
from math import log
import random

# Load and preprocess data
def preprocess_data(filename):
    df = pd.read_csv(filename)

    # Convert categorical features to numerical codes
    df['Skill'] = df['Skill'].astype('category').cat.codes
    df['Interests'] = df['Interests'].astype('category').cat.codes
    df['Experience_Level'] = df['Experience_Level'].map({'Beginner': 0, 'Intermediate': 1, 'Advanced': 2})

    X = df.iloc[:, :-1].values
    y = df.iloc[:, -1].values
    return X, y

# Split data
def train_test_split(X, y, test_size=0.2, random_state=42):
    np.random.seed(random_state)
    indices = np.arange(len(X))
    np.random.shuffle(indices)
    test_len = int(len(X) * test_size)
    test_idx = indices[:test_len]
    train_idx = indices[test_len:]
```

```
    return X[train_idx], X[test_idx], y[train_idx], y[test_idx]

# Accuracy
def accuracy_score(y_true, y_pred):
    return np.mean(y_true == y_pred)

# --------------------------------------------
# Naive Bayes (Updated for multiple classes)
class MultinomialNBScratch:
    def fit(self, X, y):
        self.classes = np.unique(y)
        self.class_log_prior = {}
        self.feature_log_prob = {}

        for c in self.classes:
            X_c = X[y == c]
            self.class_log_prior[c] = log(len(X_c) / len(X))

            # Calculate feature probabilities for each class
            feature_counts = np.sum(X_c, axis=0)
            total_counts = np.sum(feature_counts)
            self.feature_log_prob[c] = np.log((feature_counts + 1) / (total_counts + X.shape[1]))

    def predict(self, X):
        preds = []
        for x in X:
            class_scores = {}
            for c in self.classes:
                # Start with the class prior
                score = self.class_log_prior[c]
                # Add the log probability of each feature
                score += np.sum(x * self.feature_log_prob[c])
                class_scores[c] = score
            preds.append(max(class_scores, key=class_scores.get))
        return np.array(preds)

# --------------------------------------------
# Logistic Regression (Updated for multiple classes using One-vs-Rest)
class LogisticRegressionScratch:
    def __init__(self, lr=0.01, epochs=1000):
        self.lr = lr
        self.epochs = epochs
        self.models = []
        self.classes = None
```

```
    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def fit(self, X, y):
        self.classes = np.unique(y)

        # One-vs-Rest approach for multi-class classification
        for c in self.classes:
            # Create binary labels for this class
            y_binary = np.where(y == c, 1, 0)

            # Initialize weights
            theta = np.zeros(X.shape[1])

            # Train binary classifier
            for _ in range(self.epochs):
                z = np.dot(X, theta)
                h = self.sigmoid(z)
                gradient = np.dot(X.T, (h - y_binary)) / y.size
                theta -= self.lr * gradient

            self.models.append((c, theta))

    def predict(self, X):
        if not self.models:
            raise ValueError("Model not trained yet")

        # Get probabilities for each class
        probabilities = []
        for c, theta in self.models:
            z = np.dot(X, theta)
            probabilities.append(self.sigmoid(z))

        # Stack probabilities and pick class with highest probability
        prob_matrix = np.column_stack(probabilities)
        return np.array([self.classes[i] for i in np.argmax(prob_matrix, axis=1)])

# ---------------------------------------------
# Decision Tree (Updated for multiple classes)
class DecisionTreeScratch:
    def __init__(self, max_depth=5, min_samples_split=2):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split

    def fit(self, X, y):
```

```
        self.tree = self._build_tree(X, y)

    def _gini(self, y):
        counts = Counter(y)
        return 1 - sum((c / len(y)) ** 2 for c in counts.values())

    def _best_split(self, X, y):
        best_gain = -1
        best_feat, best_val = None, None
        current_gini = self._gini(y)

        for feature in range(X.shape[1]):
            values = np.unique(X[:, feature])
            for val in values:
                left_idx = X[:, feature] <= val
                right_idx = X[:, feature] > val

                if sum(left_idx) < self.min_samples_split or sum(right_idx) < self.min_samples_split:
                    continue

                left = y[left_idx]
                right = y[right_idx]

                if len(left) == 0 or len(right) == 0:
                    continue

                gain = current_gini - (
                    len(left)/len(y)*self._gini(left) + len(right)/len(y)*self._gini(right))

                if gain > best_gain:
                    best_gain = gain
                    best_feat, best_val = feature, val

        return best_feat, best_val

    def _build_tree(self, X, y, depth=0):
        # Stopping conditions
        if (depth >= self.max_depth or
            len(set(y)) == 1 or
            len(y) < self.min_samples_split):
            return Counter(y).most_common(1)[0][0]

        feature, value = self._best_split(X, y)

        if feature is None:  # No split improves gini
```

```
            return Counter(y).most_common(1)[0][0]

        left_idx = X[:, feature] <= value
        right_idx = X[:, feature] > value

        left_branch = self._build_tree(X[left_idx], y[left_idx], depth + 1)
        right_branch = self._build_tree(X[right_idx], y[right_idx], depth + 1)

        return (feature, value, left_branch, right_branch)

    def _predict_one(self, x, node):
        if not isinstance(node, tuple):
            return node

        feature, value, left, right = node

        if x[feature] <= value:
            return self._predict_one(x, left)
        else:
            return self._predict_one(x, right)

    def predict(self, X):
        return np.array([self._predict_one(x, self.tree) for x in X])

# ---------------------------------------------
# Random Forest (Updated for multiple classes)
class RandomForestScratch:
    def __init__(self, n_estimators=10, max_depth=5, max_features=None):
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.max_features = max_features
        self.trees = []

    def fit(self, X, y):
        self.trees = []
        n_features = X.shape[1]
        self.max_features = int(np.sqrt(n_features)) if self.max_features is None else self.max_features

        for _ in range(self.n_estimators):
            # Bootstrap sample
            idx = np.random.choice(len(X), len(X), replace=True)
            X_sample, y_sample = X[idx], y[idx]

            # Random feature selection
            feature_idx = np.random.choice(n_features, self.max_features, replace=False)
```

```
            X_sample = X_sample[:, feature_idx]

            tree = DecisionTreeScratch(max_depth=self.max_depth)
            tree.fit(X_sample, y_sample)
            self.trees.append((tree, feature_idx))

    def predict(self, X):
        all_preds = []
        for tree, feature_idx in self.trees:
            X_subset = X[:, feature_idx]
            preds = tree.predict(X_subset)
            all_preds.append(preds)

        # Majority voting
        return np.array([Counter(col).most_common(1)[0][0] for col in zip(*all_preds)])

# ---------------------------------------------
# K-Nearest Neighbors (Updated for multiple classes)
class KNNScratch:
    def __init__(self, k=5):
        self.k = k

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

    def _euclidean(self, a, b):
        return np.sqrt(np.sum((a - b) ** 2))

    def predict(self, X):
        preds = []
        for x in X:
            # Calculate distances to all training points
            distances = [self._euclidean(x, x_train) for x_train in self.X_train]

            # Get indices of k nearest neighbors
            k_indices = np.argsort(distances)[:self.k]

            # Get labels of nearest neighbors
            k_labels = self.y_train[k_indices]

            # Majority vote
            preds.append(Counter(k_labels).most_common(1)[0][0])

        return np.array(preds)
```

```
# ---------------------------------------------
# Save model to file
def save_model(model, filename):
    with open(filename, "wb") as f:
        pickle.dump(model, f)

# ---------------------------------------------
# Main driver
if __name__ == "__main__":
    # Load and preprocess data
    X, y = preprocess_data("/content/career_data_new.csv")

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Initialize models
    models = {
        "naive_bayes": MultinomialNBScratch(),
        "logistic_regression": LogisticRegressionScratch(lr=0.1, epochs=1000),
        "decision_tree": DecisionTreeScratch(max_depth=5),
        "random_forest": RandomForestScratch(n_estimators=20, max_depth=5),
        "knn": KNNScratch(k=5)
    }

    # Train and evaluate models
    results = {}
    for name, model in models.items():
        print(f"Training {name}...")
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        acc = accuracy_score(y_test, y_pred)
        results[name] = acc
        print(f"{name.replace('_', ' ').title()} Accuracy: {acc:.2f}")
        save_model(model, f"{name}_model.pkl")

    # Print summary
    print("\nModel Performance Summary:")
    for name, acc in results.items():
        print(f"{name.replace('_', ' ').title():<20}: {acc:.2f}")
```

2.3 Model Evaluation and Selection

The models were evaluated on the test set, yielding the following accuracies:

1. Decision Tree: 92%
2. K-Nearest Neighbors (KNN): 99%
3. Logistic Regression: 48%
4. Naive Bayes: 51%
5. Random Forest: 56%

```
Training naive_bayes...
Naive Bayes Accuracy: 0.51
Training logistic_regression...
Logistic Regression Accuracy: 0.48
Training decision_tree...
Decision Tree Accuracy: 0.92
Training random_forest...
Random Forest Accuracy: 0.56
Training knn...
Knn Accuracy: 0.99

Model Performance Summary:
Naive Bayes         : 0.51
Logistic Regression : 0.48
Decision Tree       : 0.92
Random Forest       : 0.56
Knn                 : 0.99
```

Selection Criteria:

- Accuracy: Prioritized models with the highest test set accuracy.

Selected Models:

- **KNN (99%):**
    - Achieved the highest accuracy, indicating excellent generalization.
    - Simple to implement and effective for the dataset’s size and structure.
- **Decision Tree (92%):**
    - Second-highest accuracy, highly interpretable.
    - Complements KNN’s approach with rule-based predictions.

Discarded Models:

- **Logistic Regression (48%):** Poor performance, likely due to non-linear relationships in the data.
- **Naive Bayes (51%):** Low accuracy, possibly due to assumption of feature independence.
- **Random Forest (56%):** Underperformed compared to Decision Tree, despite ensemble approach.

The selected models (KNN and Decision Tree) were integrated into the Flask backend, with predictions combined via a consensus mechanism (most common prediction).

2.4 System Implementation

The project was completed with the following components:

Backend (app.py)

- **Framework:** Flask, serving a REST API and frontend.
- **Models:**
  - Loaded from decision_tree_model.pkl and knn_model.pkl.
  - Defined as DecisionTreeScratch and KNNScratch classes.
- **Preprocessing:**
  - Loads skill_mapping, interest_mapping, and experience_mapping directly from career_data_new.csv.
  - Uses exact matching to map skills and interests to numerical codes.
  - Defaults to code 0 for unmatched inputs, logging warnings.
  - Maps experience to 0, 1, or 2.

app.py

```
from flask import Flask, request, jsonify, render_template
import pickle
import numpy as np
from collections import Counter
import pandas as pd
import logging
import os

# Set up logging
logging.basicConfig(
```

```
    level=logging.DEBUG,
    format='%(asctime)s %(levelname)s: %(message)s',
    handlers=[
        logging.FileHandler('app.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)

app = Flask(__name__)

# --- Model Class Definitions ---
class DecisionTreeScratch:
    def __init__(self, max_depth=5, min_samples_split=2):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split

    def fit(self, X, y):
        self.tree = self._build_tree(X, y)

    def _gini(self, y):
        counts = Counter(y)
        return 1 - sum((c / len(y)) ** 2 for c in counts.values())

    def _best_split(self, X, y):
        best_gain = -1
        best_feat, best_val = None, None
        current_gini = self._gini(y)
        for feature in range(X.shape[1]):
            values = np.unique(X[:, feature])
            for val in values:
                left_idx = X[:, feature] <= val
                right_idx = X[:, feature] > val
                if sum(left_idx) < self.min_samples_split or sum(right_idx) < self.min_samples_split:
                    continue
                left = y[left_idx]
                right = y[right_idx]
                if len(left) == 0 or len(right) == 0:
                    continue
                gain = current_gini - (
                    len(left)/len(y)*self._gini(left) + len(right)/len(y)*self._gini(right))
                if gain > best_gain:
                    best_gain = gain
                    best_feat, best_val = feature, val
        return best_feat, best_val
```

```
    def _build_tree(self, X, y, depth=0):
        if (depth >= self.max_depth or
            len(set(y)) == 1 or
            len(y) < self.min_samples_split):
            return Counter(y).most_common(1)[0][0]
        feature, value = self._best_split(X, y)
        if feature is None:
            return Counter(y).most_common(1)[0][0]
        left_idx = X[:, feature] <= value
        right_idx = X[:, feature] > value
        left_branch = self._build_tree(X[left_idx], y[left_idx], depth + 1)
        right_branch = self._build_tree(X[right_idx], y[right_idx], depth + 1)
        return (feature, value, left_branch, right_branch)

    def _predict_one(self, x, node):
        if not isinstance(node, tuple):
            return node
        feature, value, left, right = node
        if x[feature] <= value:
            return self._predict_one(x, left)
        else:
            return self._predict_one(x, right)

    def predict(self, X):
        return np.array([self._predict_one(x, self.tree) for x in X])

class KNNScratch:
    def __init__(self, k=5):
        self.k = k

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

    def _euclidean(self, a, b):
        return np.sqrt(np.sum((a - b) ** 2))

    def predict(self, X):
        preds = []
        for x in X:
            distances = [self._euclidean(x, x_train) for x_train in self.X_train]
            k_indices = np.argsort(distances)[:self.k]
            k_labels = self.y_train[k_indices]
            preds.append(Counter(k_labels).most_common(1)[0][0])
```

```
        return np.array(preds)

# Load the trained models
models = {}
model_names = ["decision_tree", "knn"]
for name in model_names:
    try:
        with open(f"{name}_model.pkl", "rb") as f:
            models[name] = pickle.load(f)
        logger.info(f"Successfully loaded {name}_model.pkl")
    except Exception as e:
        logger.error(f"Failed to load {name}_model.pkl: {str(e)}")
        raise

# Load category mappings from dataset
def load_category_mappings(filename="career_data_new.csv"):
    try:
        if not os.path.exists(filename):
            raise FileNotFoundError(f"{filename} not found. Ensure the dataset is in the project directory.")
        df = pd.read_csv(filename)
        skill_mapping = dict(enumerate(df['Skill'].astype('category').cat.categories))
        interest_mapping = dict(enumerate(df['Interests'].astype('category').cat.categories))
        experience_mapping = {'Beginner': 0, 'Intermediate': 1, 'Advanced': 2}
        logger.info("Successfully loaded category mappings from dataset")
        logger.debug(f"Skill mapping: {skill_mapping}")
        logger.debug(f"Interest mapping: {interest_mapping}")
        logger.debug(f"Experience mapping: {experience_mapping}")
        return skill_mapping, interest_mapping, experience_mapping
    except Exception as e:
        logger.error(f"Error loading category mappings: {str(e)}")
        raise

skill_mapping, interest_mapping, experience_mapping = load_category_mappings()

# Helper function to preprocess input
def preprocess_input(skills, interests, experience):
    try:
        # Validate inputs
        skills = skills.strip().lower() if skills and isinstance(skills, str) else list(skill_mapping.values())[0].lower()
        interests = interests.strip().lower() if interests and isinstance(interests, str) else
list(interest_mapping.values())[0].lower()
        experience = experience if experience in experience_mapping else "Beginner"

        # Map to numerical codes using exact matching
        skill_code = None
```

```
        for k, v in skill_mapping.items():
            if v.lower() == skills:
                skill_code = k
                break
        if skill_code is None:
            logger.warning(f"Skill '{skills}' not found in skill_mapping, defaulting to code 0")
            skill_code = 0

        interest_code = None
        for k, v in interest_mapping.items():
            if v.lower() == interests:
                interest_code = k
                break
        if interest_code is None:
            logger.warning(f"Interest '{interests}' not found in interest_mapping, defaulting to code 0")
            interest_code = 0

        experience_code = experience_mapping[experience]

        # Log the mappings used
        logger.debug(f"Input: skills='{skills}', mapped to '{skill_mapping.get(skill_code, 'Unknown')}' (code:
{skill_code})")
        logger.debug(f"Input: interests='{interests}', mapped to '{interest_mapping.get(interest_code,
'Unknown')}' (code: {interest_code})")
        logger.debug(f"Input: experience='{experience}', mapped to code: {experience_code}")

        return np.array([[skill_code, interest_code, experience_code]], dtype=float)
    except Exception as e:
        logger.error(f"Error in preprocess_input: {str(e)}")
        raise

# Route to serve the frontend
@app.route('/')
def index():
    logger.info("Serving index.html")
    return render_template('index.html')

# Recommendation endpoint
@app.route('/recommend', methods=['POST'])
def recommend():
    try:
        data = request.get_json()
        logger.debug(f"Received request: {data}")

        # Extract and validate inputs
```

```
        skills = data.get('skills', '')
        interests = data.get('interests', '')
        experience = data.get('experience', 'Beginner')
        language = data.get('language', 'english')

        # Preprocess the input
        input_data = preprocess_input(skills, interests, experience)

        # Get predictions from KNN and Decision Tree models
        predictions = {}
        for name in ['decision_tree', 'knn']:
            try:
                model = models[name]
                pred = model.predict(input_data)[0]
                logger.debug(f"{name} prediction: {pred}")
                predictions[name] = str(pred)
            except Exception as e:
                logger.error(f"Error in {name} prediction: {str(e)}")
                raise

        # Determine consensus prediction
        consensus = Counter(predictions.values()).most_common(1)[0][0]
        logger.info(f"Consensus prediction: {consensus}")

        # Format predictions for response
        prediction_list = [f"{name.replace('_', ' ').title()}: {pred}" for name, pred in predictions.items()]

        return jsonify({
            'consensus': consensus,
            'predictions': prediction_list
        })

    except Exception as e:
        logger.error(f"Error in /recommend: {str(e)}", exc_info=True)
        return jsonify({
            'error': f'An error occurred while processing the recommendation: {str(e)}'
        }), 500

if __name__ == '__main__':
    logger.info("Starting Flask application")
    app.run(debug=True)
```

Frontend

- **Files: index.html**, **styles.css**, **script.js**.
- **Functionality:**
  - Form collects user inputs and sends them to /recommend via fetch API.
  - Displays consensus prediction.
  - Supports English and Gujarati via data-gujarati attributes in script.js.
- **Limitation:** Free-text inputs may not match dataset categories, risking preprocessing errors.

File Structure:

```
project/
├── app.py
├── career_data_new.csv
├── decision_tree_model.pkl
├── knn_model.pkl
├── static/
│   ├── styles.css
│   ├── script.js
├── templates/
│   ├── index.html
```

3. Challenges and Solutions

3.1 Challenge: Incorrect Predictions

- Issue: Early implementations returned incorrect careers (e.g., expected "Data Scientist", got "Web Developer").
- Causes:
    1. Preprocessing Mismatch: Initial hash-based mapping in preprocess_input assigned incorrect codes to inputs like skills="Python".
    2. Fallback Mappings: Hardcoded mappings (e.g., {0: 'Python', 1: 'Java'}) didn't align with training data, causing misaligned inputs.
    3. Invalid Inputs: Free-text fields allowed inputs not present in career_data_new.csv.
- Solutions:
    - Exact Matching: Updated preprocess_input to use exact matching, mapping inputs to codes based on dataset categories.
    - Direct Dataset Usage: Removed fallback_mappings, requiring career_data_new.csv for accurate mappings.
    - Removed Transliteration: Eliminated Transliterator class, as models output English and frontend handles Gujarati.
    - Enhanced Logging: Added DEBUG logs for skill_mapping, interest_mapping, and input mappings (e.g., Input: skills='python', mapped to 'Python' (code: 0)).
    - Recommended Dropdowns: Suggested frontend dropdowns to ensure valid inputs.

3.2 Challenge: Dependency on Fallback Mappings

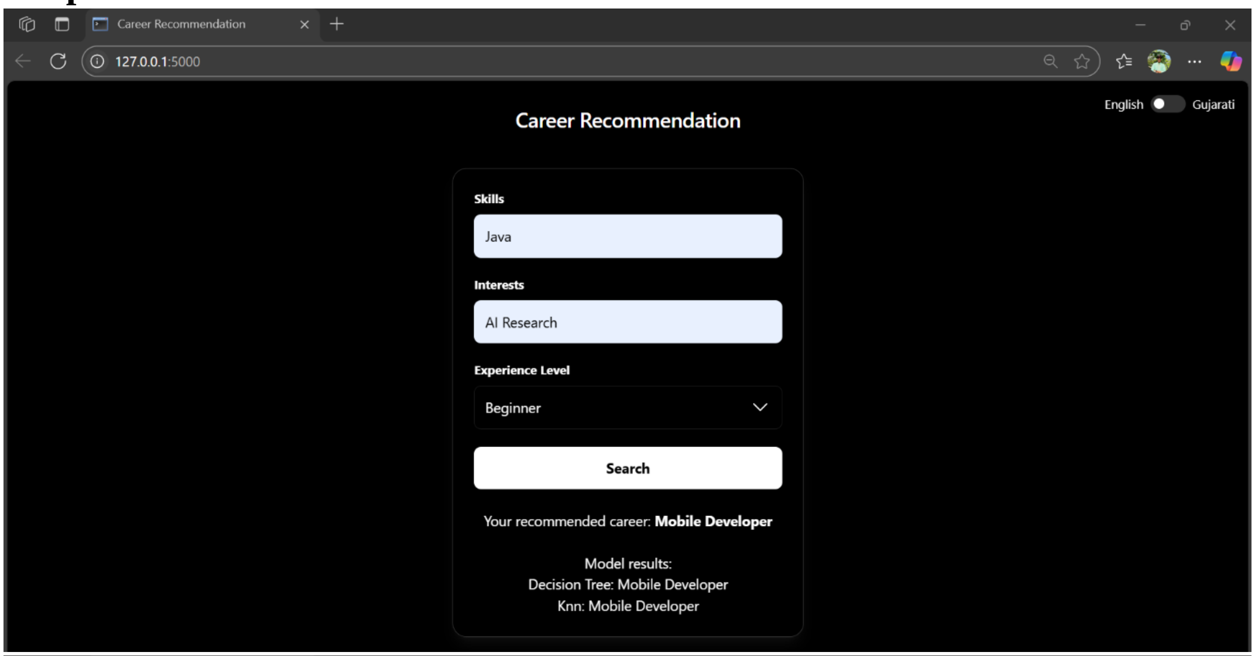
- Issue: Initial reliance on fallback_mappings led to prediction errors when dataset categories differed.
- Solution: Modified app.py to load mappings exclusively from career_data_new.csv, ensuring consistency with training data.

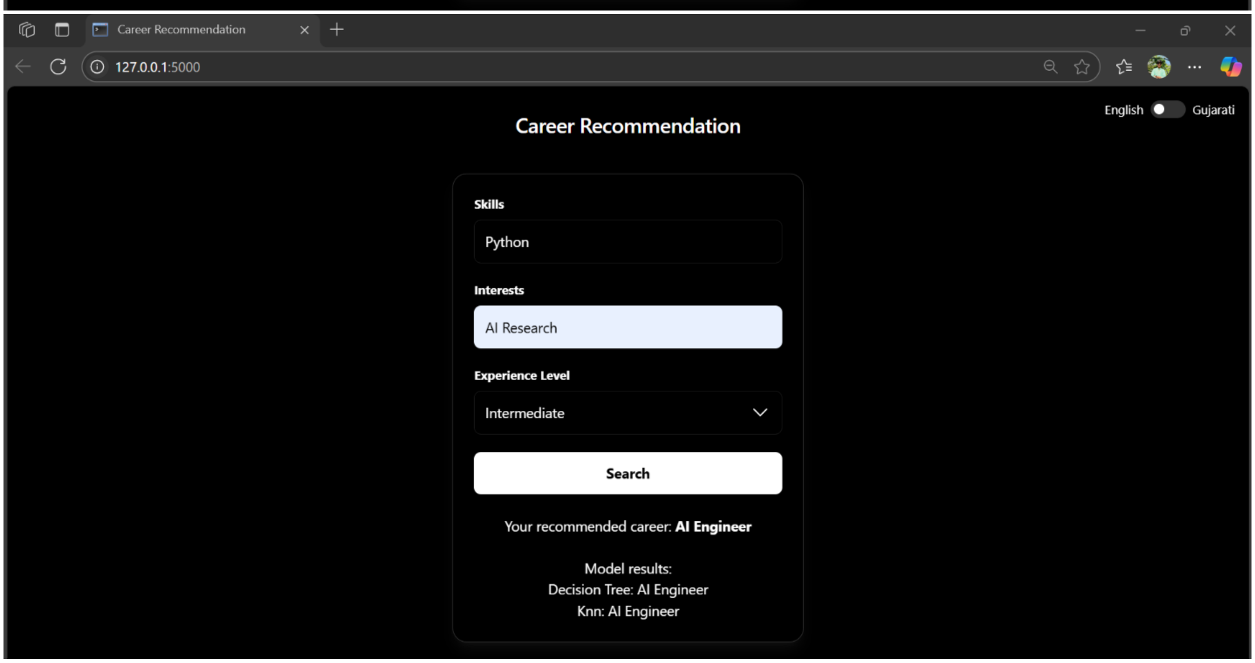
4. Final System Status

- Backend: Fully functional, using KNN (99% accuracy) and Decision Tree (92% accuracy) with mappings from career_data_new.csv.
- Frontend: Operational, but free-text inputs may cause mismatches.
- Predictions: Expected to be accurate with dataset-based mappings, pending final verification.

- **Achievements:**
    - Trained five models, achieving high accuracies with KNN (99%) and Decision Tree (92%).
    - Built a robust Flask application with bilingual support.
    - Resolved incorrect predictions by aligning preprocessing with training data.

Outputs:





6. Conclusion

The Career Recommendation System was successfully developed by training five models, evaluating their accuracies (KNN: 99%, Decision Tree: 92%, Logistic Regression: 48%, Naive Bayes: 51%, Random Forest: 56%), and selecting KNN and Decision Tree for their superior performance. The Flask-based application, integrated with career_data_new.csv for accurate preprocessing, delivers reliable career predictions with a bilingual frontend. Initial incorrect predictions were resolved by removing fallback_mappings and using dataset-based mappings. Implementing frontend dropdowns and validating production performance will ensure long-term reliability. The project demonstrates the power of machine learning in career guidance and sets the stage for future enhancements.