CSCI316: Big Data Mining Techniques and Implementation

Comprehensive Exam Notes

Part I: k-Nearest Neighbors (kNN) Algorithm

1. Overview and Concept

- **Definition**: Predicts the label of a record based on its k nearest neighbors
- **Assumption**: We already know the labels of neighboring data points
- **Principle**: Similar data points should have similar labels
- **Classification Method**: Non-parametric, lazy learning algorithm

2. kNN Algorithm Pseudocode

```
For every point in our dataset:
  1. Calculate the distance between input (inX) and the current point
  2. Sort the distances in increasing order
  3. Take k items with lowest distances to inX
  4. Find the majority class among these k items
  5. Return the majority class as prediction for inX
```

3. Distance Calculation

Euclidean Distance Formula:

- For two records A = (a_1, ..., a_n) and B = (b_1, ..., b_n):
- $|A - B| = \sqrt{[(a_1 - b_1)^2 + ... + (a_n - b_n)^2]}$

Example: Distance between "California Man" (3, 104) and "?" (18, 90) = 20.5

4. Key NumPy Functions for kNN

python

```python
import numpy as np

# Useful functions:
np.argsort(data)         # Returns indices that would sort array
np.tile(data, (2,1))      # Repeat array in specified dimensions
array.sum(axis=1)         # Sum along axis
array.shape[0]           # Get number of rows
```

5. kNN Implementation in Python

python

```python
from numpy import *

def classify0(inX, dataSet, labels, k):
    dataSetSize = dataSet.shape[0]
    # Calculate distances
    diffMat = tile(inX, (dataSetSize, 1)) - dataSet
    sqDiffMat = diffMat ** 2
    sqDistances = sqDiffMat.sum(axis=1)
    distances = sqDistances ** 0.5

    # Sort and find k nearest neighbors
    sortedDistIndicies = distances.argsort()
    classCount = {}

    # Count votes from k nearest neighbors
    for i in range(k):
        voteIlabel = labels[sortedDistIndicies[i]]
        classCount[voteIlabel] = classCount.get(voteIlabel, 0) + 1

    # Return majority class
    sortedClassCount = sorted(classCount.items(),
                        key=lambda x: x[1], reverse=True)
    return sortedClassCount[0][0]
```

6. kNN Characteristics

- **Advantages**: Simple, intuitive, effective for small datasets
- **Disadvantages**: Poor scalability for large datasets, computationally expensive
- **Use Cases**: Good as introductory algorithm, classification problems

Part II: End-to-End Machine Learning Project

1. Essential Libraries

- **Pandas**: High-performing data structure and analysis tools
  - DataFrame: 2D structure (like SQL table/spreadsheet)
- **Scikit-Learn**: Leading ML library with common algorithms
  - Works seamlessly with Pandas DataFrames

2. Eight Steps of Real-life Data Mining Project

Step 1: Look at the Big Picture

Project Example: California Housing Price Prediction (1990 Census Data)

- **Data**: Population, median income, median housing price per block group
- **Goal**: Predict median housing price for any district
- **Important**: ML model is rarely the end goal - usually part of larger system

Key Considerations:

- What is the business objective?
- How will the model be used?
- What performance measures are appropriate?
- How much effort should be invested?

Step 2: Frame the Problem

Problem Type Classification:

- **Supervised Learning**: Training examples are labeled
- **Regression Problem**: Predicting continuous values (house prices)
- **Univariate Regression**: Single target variable

Performance Measures:

- **RMSE (Root Mean Square Error)**: $RMSE(X,h) = \sqrt{[1/m \times \Sigma(h(x_i) - y_i)^2]}$
- **MAE (Mean Absolute Error)**: $MAE(X,h) = 1/m \times \Sigma|h(x_i) - y_i|$

Data Pipelines:

- Sequence of data processing components
- Components run asynchronously
- Interface between components is data store
- Makes system robust and modular

Step 3: Get the Data

python

```python
import pandas as pd
housing = pd.read_csv("house.csv")
housing.info()  # Get basic information about dataset
```

Step 4: Discover and Visualize Data

Data Exploration:

python

```python
# Create histograms for all numerical attributes
housing.hist(bins=50, figsize=(20,15))

# Geographical visualization
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)

# Complex visualization with multiple variables
housing.plot(kind="scatter", x="longitude", y="latitude",
             alpha=0.4, s=housing["population"]/100,
             c="median_house_value", cmap=plt.get_cmap("jet"),
             colorbar=True)
```

Correlation Analysis:

python

```python
# Calculate correlation matrix
corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)

# Scatter matrix for key attributes
from pandas.plotting import scatter_matrix
attributes = ["median_house_value", "median_income",
              "total_rooms", "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```

Feature Engineering:

python

```python
# Create new meaningful attributes
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"] = housing["population"]/housing["households"]
```

Step 5: Create Test Data

Two Sampling Methods:

1. Random Sampling:

python

```python
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]

train_set, test_set = split_train_test(housing, 0.2)
```

2. Stratified Sampling:

python

```python
# Create income categories for stratification
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])

# Perform stratified sampling
from sklearn.model_selection import StratifiedShuffleSplit
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

Why Stratified Sampling?

- Ensures test set is representative of overall population
- Divides population into homogeneous subgroups (strata/bins)
- Samples appropriate number from each stratum

Step 6: Prepare Data for ML Algorithms

Data Cleaning - Handling Missing Values: Three options for missing data:

1. Remove corresponding records: `housing.dropna(subset=["total_bedrooms"])`
2. Remove entire attribute: `housing.drop("total_bedrooms", axis=1)`
3. Fill with value: `housing["total_bedrooms"].fillna(median, inplace=True)`

Using SimpleImputer:

python

```python
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
housing_num = housing.drop("ocean_proximity", axis=1)  # Remove text attribute
imputer.fit(housing_num)
X = imputer.transform(housing_num)
housing_tr = pd.DataFrame(X, columns=housing_num.columns)
```

Handling Categorical Features:

```python
# Ordinal Encoding (categories as scalars)
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)

# One-Hot Encoding (categories as vectors)
from sklearn.preprocessing import OneHotEncoder
cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

Custom Transformers:

```python
from sklearn.base import BaseEstimator, TransformerMixin

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room=True):
        self.add_bedrooms_per_room = add_bedrooms_per_room

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household,
                        population_per_household, bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]
```

Feature Scaling: Two main methods:

1. **Min-Max Scaling (Normalization)**: Scale to [0,1] range
   - Use `MinMaxScaler` from Scikit-Learn
2. **Standardization**: Zero mean, unit variance
   - Use `StandardScaler` from Scikit-Learn

Transformation Pipelines:

python

```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# Numerical pipeline
num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler())
])

# Full pipeline with categorical features
from sklearn.compose import ColumnTransformer
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", OneHotEncoder(), cat_attribs)
])

housing_prepared = full_pipeline.fit_transform(housing)
```

Step 7: Select and Train Models

Model Training Examples:

1. Linear Regression:

python

```python
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)

# Make predictions
predictions = lin_reg.predict(some_data_prepared)
```

2. Decision Tree:

python

```python
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

3. Random Forest:

python

```python
from sklearn.ensemble import RandomForestRegressor
forest_reg = RandomForestRegressor()
forest_reg.fit(housing_prepared, housing_labels)
```

Model Evaluation:

python

```python
from sklearn.metrics import mean_squared_error

# Calculate RMSE
predictions = model.predict(housing_prepared)
mse = mean_squared_error(housing_labels, predictions)
rmse = np.sqrt(mse)
```

Cross-Validation:

python

```python
from sklearn.model_selection import cross_val_score

# K-fold cross-validation
scores = cross_val_score(model, housing_prepared, housing_labels,
                        scoring="neg_mean_squared_error", cv=10)
rmse_scores = np.sqrt(-scores)

def display_scores(scores):
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())
```

Important Concept: Overfitting

- When model performs perfectly on training data (RMSE = 0) but poorly on new data

- Decision tree showed this behavior in the example
- Cross-validation helps detect overfitting

Step 8: Fine-Tune Your Model

Grid Search for Hyperparameter Tuning:

```python
from sklearn.model_selection import GridSearchCV

# Define parameter grid
param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]}
]

# Perform grid search
forest_reg = RandomForestRegressor()
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                          scoring='neg_mean_squared_error',
                          return_train_score=True)
grid_search.fit(housing_prepared, housing_labels)

# Get best parameters
best_params = grid_search.best_params_
```

Final Model Evaluation:

```python
# Use best model on test set
final_model = grid_search.best_estimator_
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()
X_test_prepared = full_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_prepared)
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
```

3. Launch, Monitor and Maintain System

Production Deployment:

- Integrate with existing data sources
- Set up data pipelines
- Configure system interfaces

Monitoring Requirements:

- **Model Performance**: Track RMSE and other metrics
- **Runtime Performance**: Monitor execution time
- **Input Data Quality**: Validate incoming data

Maintenance Tasks:

- **Regular Retraining**: Update model with new data
- **Version Management**: Maintain working vs. updating versions
- **Online vs. Offline Training**: Choose appropriate update strategy

---

Key Exam Topics Summary

Critical Concepts to Remember:

1. **kNN Algorithm**: Distance calculation, majority voting, scalability issues
2. **ML Project Workflow**: All 8 steps and their purposes
3. **Data Preprocessing**: Handling missing values, categorical encoding, feature scaling
4. **Model Evaluation**: Cross-validation, overfitting detection, performance metrics
5. **Hyperparameter Tuning**: Grid search methodology
6. **Production Considerations**: Monitoring, maintenance, data pipelines

Common Pitfalls:

- Using test data during training process
- Ignoring data scaling requirements
- Not handling missing values properly
- Overfitting without cross-validation
- Forgetting about categorical variable encoding

Performance Metrics:

- **RMSE**: $\sqrt{[1/m \times \Sigma(\text{predicted - actual})^2]}$

- **MAE**: $1/m \times \Sigma|\text{predicted} - \text{actual}|$
- Cross-validation scores for robust evaluation

Python Libraries Hierarchy:

- **NumPy**: Fundamental array operations
- **Pandas**: Data manipulation and analysis
- **Scikit-Learn**: Machine learning algorithms and preprocessing
- **Matplotlib**: Data visualization