CSCI E-89B Introduction to Natural Language Processing

Harvard Extension School

Dmitry Kurochkin

Fall 2025
Lecture 9

Contents

Classical Methods for Natural Language Processing (NLP)

- **Naive Bayes Classifiers:**
  - Based on Bayes' Theorem with feature independence assumption.
  - Often used for spam detection and sentiment analysis.
  - Model is less prone to overfitting with small datasets compared to more complex models.
- **k-Nearest Neighbors (KNN):**
  - Classifies based on proximity, using distance metrics like Euclidean or cosine similarity.
  - Simple to implement and adapt, with no formal training phase.
  - Useful for smaller datasets and tasks needing interpretability, like recommendations.
- **Logistic Regression:**
  - Useful for binary text classification tasks like spam detection.
  - Uses a logistic function to convert linear feature combinations into class probabilities, similar to a single neuron in a neural network.
  - Coefficients provide interpretable insights into feature impact, in contrast to neural networks which often act as black boxes.

Classical Methods for NLP (Continued)

- **Support Vector Machines (SVM):**
  - Maximizes the margin between classes to enhance generalization.
  - Effective in high-dimensional spaces, ideal for text classification.
  - Uses kernel trick to handle non-linear relationships and complex boundaries.
- **Decision Trees:**
  - Use a tree-like structure to represent decisions and possible outcomes by splitting data based on feature values.
  - Easy to interpret and visualize, providing clear insights into the decision-making process.
  - Prone to overfitting if not properly pruned, as they can closely fit training data and capture noise as patterns.
- **Random Forests:**
  - Combine multiple decision trees to enhance prediction accuracy and reduce overfitting, creating a robust ensemble model.
  - Employ the bagging technique (Bootstrap Aggregating) to ensure diversity by training each tree on random data subsets.
  - Versatile and effective for both classification and regression tasks, handling datasets with noise or incomplete data efficiently.

Contents

Naive Bayes Classifier

- **Bayes' Theorem:**

$$P(C_k|\mathbf{x}) = \frac{P(\mathbf{x}|C_k) \cdot P(C_k)}{P(\mathbf{x})}$$

  - $P(C_k|\mathbf{x})$: Posterior probability of class C_k given features $\mathbf{x}$.
  - $P(\mathbf{x}|C_k)$: Likelihood of features $\mathbf{x}$ given class C_k.
  - $P(C_k)$: Prior probability of class C_k, computed as the number of instances in class C_k divided by the total number of instances in the training dataset.
  - $P(\mathbf{x})$: Evidence, probability of features $\mathbf{x}$.

- **Naive Independence Assumption:**

$$P(\mathbf{x}|C_k) = \prod_{i=1}^{n} P(x_i|C_k)$$

  - Assumes features $x_1, x_2, \ldots, x_n$ are conditionally independent given the class.

Naive Bayes Classifier (Continued)

- **Classification Rule:**

$$\hat{C} = \arg\max_{C_k} P(C_k|\mathbf{x})$$

  - Choose the class $\hat{C}$ with the highest posterior probability.

- **Practical Implementation:**
  - Use log probabilities to prevent numerical underflow:

$$\log P(C_k|\mathbf{x}) = \log\left(\frac{P(\mathbf{x}|C_k) \cdot P(C_k)}{P(\mathbf{x})}\right)$$

$$\propto \log P(C_k) + \sum_{i=1}^{n} \log P(x_i|C_k)$$

Naive Bayes Classifier

- **Foundation:**
  - Based on Bayes' Theorem, assuming independence between features.
  - Computes posterior probabilities by multiplying class prior by likelihood of features.
- **Applications:**
  - Widely used in text classification tasks, such as spam detection and sentiment analysis.
  - Effective for high-dimensional datasets common in NLP.
- **Advantages:**
  - Fast and efficient, with low computational cost.
  - Interpretable and easy to implement, making it suitable for baseline models.
  - Naturally handles missing data and performs well when feature independence holds approximately.

Contents

Naive Bayes Classifier in Python

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

# Sample documents
documents = [
    "Cats purr gently and climb high trees. Chasing a mouse is fun for cats.",
    "Independent creatures, cats enjoy solitude and love their nap time.",
    "Dogs bark loudly at strangers and fetch sticks with enthusiasm.",
    "Loyal dogs accompany humans on hikes and love to chase balls.",
    "When the energetic dog spotted a squirrel, it barked energetically.",
    "A purring cat climbed the bookshelf, watching over the room.",
    "Regularly, dogs enjoy long walks, sniffing and exploring their environment.",
    "Cats meow softly and purr when content, loving to stretch in the sun."
]

# Corresponding labels (0 for cat-related, 1 for dog-related)
labels = [0, 0, 1, 1, 1, 0, 1, 0]

# Split the documents and labels into training and testing sets
docs_train, docs_test, y_train, y_test = train_test_split(documents, labels,
                                                          test_size=0.25, random_state=1642)

# Vectorize the training data
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(docs_train)

# Transform the test data using the fitted vectorizer
X_test = vectorizer.transform(docs_test)
docs_test
['Dogs bark loudly at strangers and fetch sticks with enthusiasm.',
 'Cats meow softly and purr when content, loving to stretch in the sun.']
```

Naive Bayes Classifier in Python (Continued)

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score

# Train a Naive Bayes classifier
model = MultinomialNB()
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Calculate and print accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display test documents with predicted labels
for doc, prediction in zip(docs_test, y_pred):
    label_name = "Cat" if prediction == 0 else "Dog"
    print(f'"{doc}" -- Predicted: {label_name}')
```

```
Accuracy: 1.00
"Dogs bark loudly at strangers and fetch sticks with enthusiasm." -- Predicted: Dog
"Cats meow softly and purr when content, loving to stretch in the sun." -- Predicted: Cat
```

Contents

k-Nearest Neighbors (KNN)

- **Concept:**
  - A non-parametric, instance-based learning algorithm used for classification and regression.
  - Classifies data points based on the majority class among the k nearest neighbors in the feature space.
- **Distance Metrics:**
  - Commonly uses Euclidean distance, but other metrics like Manhattan, cosine similarity, and Minkowski distance can also be applied:

$$\text{Euclidean distance:} \quad d(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

$$\text{Manhattan distance:} \quad d(x,y) = \sum_{i=1}^{n}|x_i - y_i|$$

$$\text{Cosine similarity:} \quad d(x,y) = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$$

$$\text{Minkowski distance (generalized):} \quad d(x,y) = \left(\sum_{i=1}^{n}|x_i - y_i|^p\right)^{1/p}$$

KNN (Continued)

- **Classification Rule:**
  - For a given point $\mathbf{x}$, predict the class $\hat{C}$ as:

$$\hat{C} = \arg\max_{C} \sum_{i \in \mathcal{N}_k(\mathbf{x})} \mathbb{I}(y_i = C)$$

    where $\mathcal{N}_k(\mathbf{x})$ are the indices of the k nearest neighbors, and $\mathbb{I}$ is an indicator function.
- **Advantages:**
  - Simple to implement and understand.
  - Effective for large training datasets and adaptable to any metric space.
- **Limitations:**
  - Computationally expensive for large datasets.
  - Performance is sensitive to the choice of k and the distance metric.

Contents

KNN in Python

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

# Sample documents
documents = [
    "Cats purr gently and climb high trees. Chasing a mouse is fun for cats.",
    "Independent creatures, cats enjoy solitude and love their nap time.",
    "Dogs bark loudly at strangers and fetch sticks with enthusiasm.",
    "Loyal dogs accompany humans on hikes and love to chase balls.",
    "When the energetic dog spotted a squirrel, it barked energetically.",
    "A purring cat climbed the bookshelf, watching over the room.",
    "Regularly, dogs enjoy long walks, sniffing and exploring their environment.",
    "Cats meow softly and purr when content, loving to stretch in the sun."
]

# Corresponding labels (0 for cat-related, 1 for dog-related)
labels = [0, 0, 1, 1, 1, 0, 1, 0]

# Split the documents and labels into training and testing sets
docs_train, docs_test, y_train, y_test = train_test_split(documents, labels,
                                                          test_size=0.25, random_state=1642)

# Vectorize the training data
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(docs_train)

# Transform the test data using the fitted vectorizer
X_test = vectorizer.transform(docs_test)
docs_test
['Dogs bark loudly at strangers and fetch sticks with enthusiasm.',
 'Cats meow softly and purr when content, loving to stretch in the sun.']
```

KNN in Python (Continued)

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Initialize the k-NN classifier with a chosen number of neighbors, e.g., k=3
knn = KNeighborsClassifier(n_neighbors=5)

# Train the k-NN classifier
knn.fit(X_train, y_train)

# Make predictions on the test set
y_pred = knn.predict(X_test)

# Calculate and print accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy with k-NN: {accuracy:.2f}')

# Display test documents with predicted labels
for doc, prediction in zip(docs_test, y_pred):
    label_name = "Cat" if prediction == 0 else "Dog"
    print(f'"{doc}" -- Predicted: {label_name}')
```

```
Accuracy with k-NN: 1.00
"Dogs bark loudly at strangers and fetch sticks with enthusiasm." -- Predicted: Dog
"Cats meow softly and purr when content, loving to stretch in the sun." -- Predicted: Cat
```

Contents

Logistic Regression

- **Concept:**
  - A linear model used for binary classification tasks, modeling the probability of class membership.
  - Uses the logistic (sigmoid) function to map linear combinations of features to probability scores.
  - Equivalent to a single-layer neural network with one neuron, using the sigmoid activation function.

- **Mathematical Formulation:**

$$\begin{aligned} \text{Linear model:} \quad & z = \mathbf{w}^T \mathbf{x} + b \\ \text{Sigmoid function:} \quad & \sigma(z) = \frac{1}{1 + e^{-z}} \\ \text{Predicted probability:} \quad & P(y = 1|\mathbf{x}) = \sigma(z) \end{aligned}$$

Logistic Regression (Continued)

- **Applications:**
  - Widely used in text classification, image processing, and bioinformatics for tasks requiring binary outcomes.
  - Suitable for problems where interpretability and model transparency are important.
- **Advantages:**
  - Simple, efficient, and easy to implement.
  - Coefficients are interpretable, indicating the influence of each feature on the class probability.
- **Limitations:**
  - Assumes a linear relationship between features and log-odds.
  - May underperform if the decision boundary is complex or non-linear.

Contents

Logistic Regression in Python

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

# Sample documents
documents = [
    "Cats purr gently and climb high trees. Chasing a mouse is fun for cats.",
    "Independent creatures, cats enjoy solitude and love their nap time.",
    "Dogs bark loudly at strangers and fetch sticks with enthusiasm.",
    "Loyal dogs accompany humans on hikes and love to chase balls.",
    "When the energetic dog spotted a squirrel, it barked energetically.",
    "A purring cat climbed the bookshelf, watching over the room.",
    "Regularly, dogs enjoy long walks, sniffing and exploring their environment.",
    "Cats meow softly and purr when content, loving to stretch in the sun."
]

# Corresponding labels (0 for cat-related, 1 for dog-related)
labels = [0, 0, 1, 1, 1, 0, 1, 0]

# Split the documents and labels into training and testing sets
docs_train, docs_test, y_train, y_test = train_test_split(documents, labels,
                                                          test_size=0.25, random_state=1642)

# Vectorize the training data
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(docs_train)

# Transform the test data using the fitted vectorizer
X_test = vectorizer.transform(docs_test)
docs_test
['Dogs bark loudly at strangers and fetch sticks with enthusiasm.',
 'Cats meow softly and purr when content, loving to stretch in the sun.']
```

Logistic Regression in Python (Continued)

```
from sklearn.linear_model import LogisticRegression

# Initialize the Logistic Regression classifier
model = LogisticRegression(max_iter=1000)

# Train the Logistic Regression classifier
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Calculate and print accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Display test documents with predicted labels
for doc, prediction in zip(docs_test, y_pred):
    label_name = "Cat" if prediction == 0 else "Dog"
    print(f'"{doc}" -- Predicted: {label_name}')
```

```
Accuracy: 1.00
"Dogs bark loudly at strangers and fetch sticks with enthusiasm." -- Predicted: Dog
"Cats meow softly and purr when content, loving to stretch in the sun." -- Predicted: Cat
```

Contents

Support Vector Machines (SVM)

- **Concept:**
  - Supervised learning algorithm used for classification and regression.
  - Finds the optimal hyperplane that maximizes the margin between different classes in the feature space.
  - Utilizes support vectors, which are the critical data points nearest to the decision boundary.

SVM (Continued)

- **Mathematical Formulation:**
  - **Hyperplane and Margin:**

$$\mathbf{w}^T\mathbf{x} + b = 0$$

  - **Optimization Problem (Hard Margin Case):**

$$\min \frac{1}{2}\|\mathbf{w}\|^2$$
$$\text{subject to } y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$$

  - **Optimization Problem (Soft Margin Case):**

$$\min \left(\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \xi_i \right)$$
$$\text{subject to } y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0,$$

where ξ_i indicates the degree to which instance i violates the margin.

SVM (Continued)

- **Kernels:**
  - Kernel functions enable SVMs to handle non-linearly separable data by implicitly mapping input features into higher-dimensional spaces.
  - Common kernel types include:
    - ★ **Polynomial Kernel:** Captures interactions between features up to a specified degree, allowing complex decision boundaries:

$$K(x, y) = (\mathbf{x}^T \mathbf{y} + c)^d$$

      where c is a constant term and d is the degree of the polynomial.
    - ★ **Radial Basis Function (RBF) Kernel:** Also known as the Gaussian kernel, effective for capturing similarity based on distance:

$$K(x, y) = \exp\left(-\gamma \|\mathbf{x} - \mathbf{y}\|^2\right)$$

      where γ is a parameter defining the influence of points.
  - Kernels allow SVMs to create non-linear decision boundaries without explicitly computing the coordinates of data in high-dimensional space, leveraging the "kernel trick."
  - Choice of kernel impacts model complexity, accuracy, and computational efficiency, requiring careful selection and parameter tuning.

Applications of SVM

- **Text Classification:**
  - SVM is used for tasks like spam detection, sentiment analysis, and document categorization.
  - It effectively handles high-dimensional data typical in text processing.
- **Image Recognition:**
  - Applicable in handwriting recognition and object detection.
  - Utilizes kernel functions to learn complex image patterns.
- **Bioinformatics:**
  - Used for classifying proteins and disease prediction.
  - Manages large datasets with numerous variables, such as genetic data.
- **Key Features of SVM:**
  - Versatile across domains, supporting both linear and non-linear data separations.
  - Robust against overfitting with a well-tuned soft margin.
  - Highly popular before the resurgence of neural networks in the early 2010s, especially effective for smaller datasets and structured data.

Contents

SVM in Python

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

# Sample documents
documents = [
    "Cats purr gently and climb high trees. Chasing a mouse is fun for cats.",
    "Independent creatures, cats enjoy solitude and love their nap time.",
    "Dogs bark loudly at strangers and fetch sticks with enthusiasm.",
    "Loyal dogs accompany humans on hikes and love to chase balls.",
    "When the energetic dog spotted a squirrel, it barked energetically.",
    "A purring cat climbed the bookshelf, watching over the room.",
    "Regularly, dogs enjoy long walks, sniffing and exploring their environment.",
    "Cats meow softly and purr when content, loving to stretch in the sun."
]

# Corresponding labels (0 for cat-related, 1 for dog-related)
labels = [0, 0, 1, 1, 1, 0, 1, 0]

# Split the documents and labels into training and testing sets
docs_train, docs_test, y_train, y_test = train_test_split(documents, labels,
                                                          test_size=0.25, random_state=1642)

# Vectorize the training data
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(docs_train)

# Transform the test data using the fitted vectorizer
X_test = vectorizer.transform(docs_test)
docs_test
['Dogs bark loudly at strangers and fetch sticks with enthusiasm.',
 'Cats meow softly and purr when content, loving to stretch in the sun.']
```

SVM with Linear Kernel in Python

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Initialize the SVM classifier
svm = SVC(kernel='linear')  # Linear

# Train the SVM classifier
svm.fit(X_train, y_train)

# Make predictions on the test set
y_pred = svm.predict(X_test)

# Calculate and print accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy with SVM: {accuracy:.2f}')

# Display test documents with predicted labels
for doc, prediction in zip(docs_test, y_pred):
    label_name = "Cat" if prediction == 0 else "Dog"
    print(f'"{doc}" -- Predicted: {label_name}')
```

```
Accuracy with SVM: 1.00
"Dogs bark loudly at strangers and fetch sticks with enthusiasm." -- Predicted: Dog
"Cats meow softly and purr when content, loving to stretch in the sun." -- Predicted: Cat
```

SVM with Radial Basis Function Kernel in Python

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Initialize the SVM classifier
svm = SVC(kernel='rbf')  # Radial Basis Function

# Train the SVM classifier
svm.fit(X_train, y_train)

# Make predictions on the test set
y_pred = svm.predict(X_test)

# Calculate and print accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy with SVM: {accuracy:.2f}')

# Display test documents with predicted labels
for doc, prediction in zip(docs_test, y_pred):
    label_name = "Cat" if prediction == 0 else "Dog"
    print(f'"{doc}" -- Predicted: {label_name}')
```

```
Accuracy with SVM: 1.00
"Dogs bark loudly at strangers and fetch sticks with enthusiasm." -- Predicted: Dog
"Cats meow softly and purr when content, loving to stretch in the sun." -- Predicted: Cat
```

SVM with Polynomial Kernel in Python

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Initialize the SVM classifier
svm = SVC(kernel='poly')  # Polynomial Kernel

# Train the SVM classifier
svm.fit(X_train, y_train)

# Make predictions on the test set
y_pred = svm.predict(X_test)

# Calculate and print accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy with SVM: {accuracy:.2f}')

# Display test documents with predicted labels
for doc, prediction in zip(docs_test, y_pred):
    label_name = "Cat" if prediction == 0 else "Dog"
    print(f'"{doc}" -- Predicted: {label_name}')
```

```
Accuracy with SVM: 1.00
"Dogs bark loudly at strangers and fetch sticks with enthusiasm." -- Predicted: Dog
"Cats meow softly and purr when content, loving to stretch in the sun." -- Predicted: Cat
```

Contents

Random Trees and Random Forests

- **Concept:**
  - A Random Tree is a decision tree constructed by choosing splits randomly from a subset of the available features.
  - A Random Forest is an ensemble learning method that combines multiple random trees to improve predictive accuracy and control overfitting.

Random Trees and Random Forests (Continued)

- **Mathematical Formulation of a Decision Tree:**
  - Constructed using a random subset of the training data through bootstrapping (sampling with replacement).
  - At each node, a random subset of features is selected, and the best split is determined using criteria like information gain or Gini impurity:

$$\text{Entropy} = -\sum_i p_i \log_2 p_i$$

$$\text{Gini index (impurity)} = 1 - \sum_i p_i^2$$

  - The feature that results in the highest decrease in entropy or Gini impurity (or highest information gain) is used to split the node.
  - Each tree independently makes predictions, and these are aggregated in the Random Forest for final decisions.

Random Trees and Random Forests (Continued)

- **Mathematical Formulation of a Random Forest:**
  - **Concept:**
    - ★ Random Forests are an ensemble learning method that combine predictions from multiple decision trees, improving model performance by reducing variance and enhancing generalization.
    - ★ Each decision tree is constructed from bootstrapped samples of the training data, with random subsets of features selected for each split, ensuring diverse and robust tree decisions that help mitigate overfitting.
  - **Aggregation Mechanism:**
    - ★ For classification, let $h_b(x)$ denote the prediction from the b^{th} decision tree, where B is the total number of trees. The prediction by the Random Forest is given by:

$$\hat{y} = \text{mode}\{h_1(x), h_2(x), \ldots, h_B(x)\}$$

    - ★ For regression, let $h_b(x)$ denote the prediction from the b^{th} decision tree, where B is the total number of trees. The prediction by the Random Forest is given by:

$$\hat{y} = \frac{1}{B}\sum_{b=1}^{B} h_b(x)$$

Random Trees and Random Forests (Continued)

- **Feature Importance:**
  - Evaluate feature importance by assessing the contribution of each feature across all trees, often using metrics like mean decrease in impurity.
- **Applications:**
  - Suitable for classification and regression tasks across various domains.
  - Commonly used in financial predictions, healthcare, and environmental modeling due to robustness and interpretability.
- **Key Features:**
  - Handles high-dimensional data well, reducing variance through ensemble averaging.
  - Offers insights into feature importance, aiding interpretability.

Contents

Random Forest in Python

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

# Sample documents
documents = [
    "Cats purr gently and climb high trees. Chasing a mouse is fun for cats.",
    "Independent creatures, cats enjoy solitude and love their nap time.",
    "Dogs bark loudly at strangers and fetch sticks with enthusiasm.",
    "Loyal dogs accompany humans on hikes and love to chase balls.",
    "When the energetic dog spotted a squirrel, it barked energetically.",
    "A purring cat climbed the bookshelf, watching over the room.",
    "Regularly, dogs enjoy long walks, sniffing and exploring their environment.",
    "Cats meow softly and purr when content, loving to stretch in the sun."
]

# Corresponding labels (0 for cat-related, 1 for dog-related)
labels = [0, 0, 1, 1, 1, 0, 1, 0]

# Split the documents and labels into training and testing sets
docs_train, docs_test, y_train, y_test = train_test_split(documents, labels,
                                                          test_size=0.25, random_state=1642)

# Vectorize the training data
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(docs_train)

# Transform the test data using the fitted vectorizer
X_test = vectorizer.transform(docs_test)
docs_test
['Dogs bark loudly at strangers and fetch sticks with enthusiasm.',
 'Cats meow softly and purr when content, loving to stretch in the sun.']
```

Random Forest in Python (Continued)

```
from sklearn.ensemble import RandomForestClassifier

# Initialize the Random Forest classifier
rf = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the Random Forest classifier
rf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = rf.predict(X_test)

# Calculate and print accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy with Random Forest: {accuracy:.2f}')

# Display test documents with predicted labels
for doc, prediction in zip(docs_test, y_pred):
    label_name = "Cat" if prediction == 0 else "Dog"
    print(f'"{doc}" -- Predicted: {label_name}')


Accuracy with Random Forest: 1.00
"Dogs bark loudly at strangers and fetch sticks with enthusiasm." -- Predicted: Dog
"Cats meow softly and purr when content, loving to stretch in the sun." -- Predicted: Cat
```