The Naive Bayes Algorithm: Takeaways

Concepts

- When a new message "w_1, w_2, ..., w_n" comes in, the Naive Bayes algorithm classifies it as spam or non-spam based on the results of these two equations:

$$P(Spam|w_1, w_2, ..., w_n) \propto P(Spam) \cdot \prod_{i=1}^{n} P(w_i|Spam)$$
$$P(Spam^C|w_1, w_2, ..., w_n) \propto P(Spam^C) \cdot \prod_{i=1}^{n} P(w_i|Spam^C)$$

- To calculate $P(w_i|Spam)$ and $P(w_i|Spam^C)$, we need to use the additive smoothing technique:

$$P(w_i|Spam) = \frac{N_{w_i|Spam} + \alpha}{N_{Spam} + \alpha \cdot N_{Vocabulary}}$$

$$P(w_i|Spam^C) = \frac{N_{w_i|Spam^C} + \alpha}{N_{Spam^C} + \alpha \cdot N_{Vocabulary}}$$

- Below, we see what some of the terms in equations above mean:

\begin{aligned} &N;_{w_i|Spam} = \text{the number of times the word } w_i \text{ occurs in spam messages} \\ &N;_{w_i|Spam^C} = \text{the number of times the word } w_i \text{ occurs in non-spam messages} \\ \\ &N;_{Spam} = \text{total number of words in spam messages} \\ &N;_{Spam^C} = \text{total number of words in non-spam messages} \\ \\ &N;_{Vocabulary} = \text{total number of words in the vocabulary} \\ &\alpha = 1 \ \ \ \ (\alpha \text{ is a smoothing parameter}) \end{aligned}

Resources

- A technical intro to a few version of the Naive Bayes algorithm
- An intro to conditional independence

Introduction to K-Nearest Neighbors: Takeaways

Syntax

- Randomizing the order of a DataFrame:

```
import numpy as np
np.random.seed(1)
np.random.permutation(len(dc_listings))
```

- Returning a new DataFrame containing the shuffled order:

```
dc_listings = dc_listings.loc[np.random.permutation(len(dc_listings))]
```

- Applying string methods to replace a comma with an empty character:

```
stripped_commas = dc_listings['price'].str.replace(',', '')
```

- Converting a Series object to a float datatype:

```
dc_listings['price'] = dc_listings['price'].astype('float')
```

Concepts

- Machine learning is the process of discovering patterns in existing data to make a prediction.
- In machine learning, a feature is an individual measurable characteristic.
- When predicting a continuous value, the main similarity metric that's used is Euclidean distance.
- K-nearest neighbors computes the Euclidean Distance to find similarity and average to predict an unseen value.
- Let q_1 to q_n represent the feature values for one observation, and p_1 to p_n represent the feature values for the other observation then the formula for Euclidean distance is as follows:

$$d = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \ldots + (q_n - p_n)^2}$$

- In the case of one feature (univariate case), the Euclidean distance formula is as follows:

$$d = \sqrt{(q_1 - p_1)^2}$$

Resources

- K-Nearest Neighbors
- Five Popular Similarity Measures

Evaluating Model Performance: Takeaways

Syntax

- Calculating the (mean squared error) MSE:

```
test_df['squared_error'] = (test_df['predicted_price'] - test_df['price'])**(2)
mse = test_df['squared_error'].mean()
```

- Calculating the (mean absolute error) MAE:

```
test_df['squared_error'] = np.absolute(test_df['predicted_price'] - test_df['price'])
mae = test_df['squared_error'].mean()
```

- Calculating the root mean squared error (RMSE):

```
test_df['squared_error'] = (test_df['predicted_price'] - test_df['price'])**(2)
mse = test_df['squared_error'].mean()
rmse = mse ** (1/2)
```

Concepts

- A machine learning model outputs a prediction based on the input to the model.
- When you're beginning to implement a machine learning model, you'll want to have some kind of validation to ensure your machine learning model can make accurate predictions on new data.
- You can test the quality of your model by:
    - Splitting the data into two sets:
        - The training set, which contains the majority of the rows (75%).
        - The test set, which contains the remaining rows (25%).
    - Using the rows in the training set to predict the values for the rows in the test set.
    - Comparing the actual values with the predicted values to see how accurate the model is.
- To quantify how good the predictions are for the test set, you would use an error metric. The error metric quantifies the difference between each predicted and actual value and then averaging those differences.
    - This is known as the mean error but isn't effective in most cases because positive and negative differences are treated differently.
- The MAE computes the absolute value of each error before we average all the errors.
- Let n be the number of observations then the MAE equation is as follows:

$$MAE = \frac{1}{n}\sum_{k=1}^{n}|(actual_1 - predicted_1)| + \ldots + |(actual_n - predicted_n)|$$

- The MSE makes the gap between the predicted and actual values more clear by squaring the difference of the two values.
- Let n be the number of observations then the MSE equation is as follows:

$$MSE = \frac{1}{n}\sum_{k=1}^{n}(actual_1 - predicted_1)^2 + \ldots + (actual_n - predicted_n)^2$$

- RMSE is an error metric whose units are the base unit, and is calculated as follows:

$$RMSE = \sqrt{\frac{1}{n}\sum_{k=1}^{n}(actual_1 - predicted_1)^2 + \ldots + (actual_n - predicted_n)^2}$$

- In general, the MAE value is expected to be be much less than the RMSE value due the sum of the squared differences before averaging.

Resources

- MAE and RMSE comparison
- About Train, Validation, and Test Sets in Machine Learning

Multivariate K-Nearest Neighbors: Takeaways

Syntax

- Displaying the number of non-null values in the columns of a DataFrame:

```
dc_listings.info()
```

- Removing rows from a DataFrame that contain a missing value:

```
dc_listings.dropna(axis=0, inplace=True)
```

- Normalizing a column using pandas:

```
first_transform = dc_listings['maximum_nights'] - dc_listings['maximum_nights'].mean()
normalized_col = first_transform / first_transform.std()
```

- Normalizing a DataFrame using pandas:

```
normalized_listings = (dc_listings - dc_listings.mean()) / (dc_listings.std())
```

- Calculating Euclidean distance using SciPy:

```
from scipy.spatial import distance
first_listing = [-0.596544, -0.439151]
second_listing = [-0.596544, 0.412923]
dist = distance.euclidean(first_listing, second_listing)
```

- Using the KNeighborsRegressor to instantiate an empty model for K-Nearest Neighbors:

```
from sklearn.neighbors import KNeighborsRegressor
knn = KNeighborsRegressor()
```

- Using the fit method to fit the K-Nearest Neighbors model to the data:

```
train_df = normalized_listings.iloc[0:2792]
test_df = normalized_listings.iloc[2792:]
train_features = train_df[['accommodates', 'bathrooms']]
train_target = train_df['price']
knn.fit(train_features, train_target)
```

- Using the predict method to make predictions on the test set:

```
predictions = knn.predict(test_df[['accommodates', 'bathrooms']])
```

- Calculating MSE using scikit-learn:

```
from sklearn.metrics import mean_squared_error
two_features_mse = mean_squared_error(test_df['price'], predictions)
```

Concepts

- To reduce the RMSE value during validation and improve accuracy, you can:
    - Select the relevant attributes a model uses. When selecting attributes, you want to make sure you're not working with a column that doesn't have continuous values. The process of selecting features to use in a model is known as feature selection.

    - Increase the value of `k` in our algorithm.
- We can normalize the columns to prevent any single value having too much of an impact on distance. Normalizing the values to a standard normal distribution preserves the distribution while aligning the scales. Let `x` be a value in a specific column, μ be the mean of all values within a single column, and σ be the standard deviation of the values within a single column, then the mathematical formula to normalize the values is as follows:
$$x = \frac{x-\mu}{\sigma}$$
- The `distance.euclidean()` function from `scipy.spatial` expects:
    - Both of the vectors to be represented using a list-like object (Python list, NumPy array, or pandas Series).
    - Both of the vectors must be 1-dimensional and have the same number of elements.
- The scikit-learn library is the most popular machine learning library in Python. Scikit-learn contains functions for all of the major machine learning algorithms implemented as a separate class. The workflow consists of four main steps:
    - Instantiate the specific machine learning model you want to use.
    - Fit the model to the training data.
    - Use the model to make predictions.
    - Evaluate the accuracy of the predictions.
- One main class of machine learning models is known as a regression model, which predicts numerical value. The other main class of machine learning models is called classification, which is used when we're trying to predict a label from a fixed set of labels.
- The fit method accepts list-like objects while the predict method accepts matrix like objects.
- The `mean_squared_error()` function takes in two inputs:
    - A list-like object representing the actual values.
    - A list like object representing the predicted values using the model.

Resources

- Scikit-learn library
- K-Neighbors Regressor

Hyperparameter Optimization: Takeaways

Syntax

- Using Grid Search to find the optimal k value:

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
cols = ['accommodates', 'bedrooms', 'bathrooms', 'number_of_reviews']
hyper_params = [x for x in range(5)]
mse_values = list()
for value in hyper_params:
    knn = KNeighborsRegressor(n_neighbors=value, algorithm='brute')
    knn.fit(train_df[cols], train_df['price'])
    predictions = knn.predict(test_df[cols])
    mse = mean_squared_error(test_df['price'], predictions)
    mse_values.append(mse)
```

- Plotting to visualize the optimal k value:

```
features = ['accommodates', 'bedrooms', 'bathrooms', 'number_of_reviews']
hyper_params = [x for x in range(1, 21)]
mse_values = list()
for hp in hyper_params:
    knn = KNeighborsRegressor(n_neighbors=hp, algorithm='brute')
    knn.fit(train_df[features], train_df['price'])
    predictions = knn.predict(test_df[features])
    mse = mean_squared_error(test_df['price'], predictions)
    mse_values.append(mse)
plt.scatter(hyper_params, mse_values)
plt.show()
```

Concepts

- Hyperparameters are values that affect the behavior and performance of a model that are unrelated to the data. Hyperparameter optimization is the process of finding the optimal hyperparameter value.
- Grid search is a simple but common hyperparameter optimization technique, which involves evaluating the model performance at different k values and selecting the k value that resulted in the lowest error. Grid search involves:
    - Selecting a subset of the possible hyperparameter values.
    - Training a model using each of these hyperparameter values.
    - Evaluating each model's performance.
    - Selecting the hyperparameter value that resulted in the lowest error value.

- The general workflow for finding the best model is:
    - Selecting relevant features to use for predicting the target column.
    - Using grid search to find the optimal hyperparameter value for the selected features.
    - Evaluate the model's accuracy and repeat the process.

Resources

- Difference Between Parameter and Hyperparameter
- Hyperparameter Optimization

Cross Validation: Takeaways

Syntax

- Implementing holdout validation:

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
train_one = split_one
test_one = split_two
train_two = split_two
test_two = split_one
model = KNeighborsRegressor()
model.fit(train_one[["accommodates"]], train_one["price"])
test_one["predicted_price"] = model.predict(test_one[["accommodates"]])
iteration_one_rmse = mean_squared_error(test_one["price"],
test_one["predicted_price"])**(1/2)
model.fit(train_two[["accommodates"]], train_two["price"])
test_two["predicted_price"] = model.predict(test_two[["accommodates"]])
iteration_two_rmse = mean_squared_error(test_two["price"],
test_two["predicted_price"])**(1/2)
avg_rmse = np.mean([iteration_two_rmse, iteration_one_rmse])
```

- Implementing k-fold cross validation:

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
model = KNeighborsRegressor()
train_iteration_one = dc_listings[dc_listings["fold"] != 1]
test_iteration_one = dc_listings[dc_listings["fold"] == 1].copy()
model.fit(train_iteration_one[["accommodates"]], train_iteration_one["price"])
labels = model.predict(test_iteration_one[["accommodates"]])
test_iteration_one["predicted_price"] = labels
iteration_one_mse = mean_squared_error(test_iteration_one["price"],
test_iteration_one["predicted_price"])
iteration_one_rmse = iteration_one_mse ** (1/2)
```

- Instantiating an instance of the KFold class from sklearn.model_selection:

```
from sklearn.model_selection import cross_val_score, KFold
kf = KFold(5, shuffle=True, random_state=1)
```

- Implementing cross_val_score along with the KFold class:

```
from sklearn.model_selection import cross_val_score
model = KNeighborsRegressor()
mses = cross_val_score(model, dc_listings[["accommodates"]], dc_listings["price"],
scoring="neg_mean_squared_error", cv=kf)
```

Concepts

- Holdout validation is a more robust technique for testing a machine learning model's accuracy on new data the model wasn't trained on. Holdout validation involves:
    - Splitting the full data set into two partitions:
        - A training set.
        - A test set.
    - Training the model on the training set.
    - Using the trained model to predict labels on the test set.
    - Computing an error to understand the model's effectiveness.
    - Switching the training and test sets and repeat.
    - Averaging the errors.
- In holdout validation, we use a 50/50 split instead of the 75/25 split from train/test validation to eliminate any sort of bias towards a specific subset of data.
- Holdout validation is a specific example of k-fold cross-validation, which takes advantage of a larger proportion of the data during training while still rotating through different subsets of the data, when `k` is set to two.
- K-fold cross-validation includes:
    - Splitting the full data set into `k` equal length partitions:
        - Selecting `k-1` partitions as the training set.
        - Selecting the remaining partition as the test set.
    - Training the model on the training set.
    - Using the trained model to predict labels on the test fold.
    - Computing the test fold's error metric.
    - Repeating all of the above steps `k-1` times, until each partition has been used as the test set for an iteration.
    - Calculating the mean of the `k` error values.
- The parameters for the KFold class are:
    - `n_splits` : The number of folds you want to use.
    - `shuffle` : Toggle shuffling of the ordering of the observations in the data set.
    - `random_state` : Specify the random seed value if `shuffle` is set to `True` .
- The parameters for using `cross_val_score` are:
    - `estimator` : Scikit-learn model that implements the `fit` method (e.g. instance of KNeighborsRegressor).
    - `X` : The list or 2D array containing the features you want to train on.
    - `y` : A list containing the values you want to predict (target column).
    - `scoring` : A string describing the scoring criteria.
    - `cv` : The number of folds. Here are some examples of accepted values:
        - An instance of the `KFold` class.
        - An integer representing the number of folds.

- The workflow for k-fold cross-validation with scikit-learn includes:
    - Instantiating the scikit-learn model class you want to fit.
    - Instantiating the KFold class and using the parameters to specify the k-fold cross-validation attributes you want.
    - Using the cross_val_score() function to return the scoring metric you're interested in.
- Bias describes error that results in bad assumptions about the learning algorithm. Variance describes error that occurs because of the variability of a model's predicted value. In an ideal world, we want low bias and low variance when creating machine learning models.

Resources

- Accepted values for scoring criteria
- Bias-variance Trade-off
- K-Fold cross-validation documentation

The Linear Regression Model: Takeaways

Syntax

- Importing and instantiating a linear regression model:

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
```

- Using the

```
LinearRegression
```

class to fit a linear regression model between a set of columns:

```
lr.fit(train[['Gr Liv Area']], train['SalePrice'])
```

- Returning the a_1 and a_0 parameters for $y = a_0 + a_1x_1$:

```
a0 = lr.intercept_
a1 = lr.coef_
```

- Predicting the labels using the training data:

```
test_predictions = lr.predict(test[['Gr Liv Area']])
```

- Calculating the correlation between pairs of columns:

```
train[['Garage Area', 'Gr Liv Area', 'Overall Cond', 'SalePrice']].corr()
```

Concepts

- An instance-based learning algorithm, such as K-nearest neighbors, relies completely on previous instances to make predictions. K-nearest neighbors doesn't try to understand or capture the relationship between the feature columns and the target column.
- Parametric machine learning, like linear regression and logistic regression, results in a mathematical function that best approximates the patterns in the training set. In machine learning, this function is often referred to as a model. Parametric machine learning approaches work by making assumptions about the relationship between the features and the target column.
- The following equation is the general form of the simple linear regression model:

$$\hat{y} = a_1x_1 + a_0$$

where $\hat{y}$ represents the target column while x_1 represents the feature column we chose to use in our model. a_0 and a_1 represent the parameter values that are specific to the dataset.

- The goal of simple linear regression is to find the optimal parameter values that best describe the relationship between the feature column and the target column.
- We minimize the model's residual sum of squares to find the optimal parameters for a linear regression model. The equation for the model's residual sum of squares is as follows:

$$RSS = (y_1 - \hat{y_1})^2 + (y_2 - \hat{y_2})^2 + \ldots + (y_n - \hat{y_n})^2$$

where $\hat{y_n}$ is our target column and y are our true values.

- A multiple linear regression model allows us to capture the relationship between multiple feature columns and the target column. The formula for multiple linear regression is as follows:

$$\hat{y} = a_0 + a_1x_1 + a_2x_2 + \ldots + a_nx_n$$

where x_1 to x_n are our feature columns, and the parameter values that are specific to the data set are represented by a_0 along with a_1 to a_n.

- In linear regression, it is a good idea to select features that are a good predictor of the target column.

Resources

- Linear Regression Documentation
- pandas.DataFrane.corr() Documentation

Feature Selection: Takeaways

Syntax

- Using Seaborn to generate a correlation matrix heatmap:

```
sns.heatmap(DataFrame)
```

- Rescaling the features for a model:

```
data = pd.read_csv('AmesHousing.txt', delimiter="\t")
train = data[0:1460]
unit_train = (train[['Gr Liv Area']] - train['Gr Liv Area'].min())/(train['Gr Liv
Area'].max() - train['Gr Liv Area'].min())
```

Concepts

- Once we select the model we want to use, selecting the appropriate features for that model is the next important step. When selecting features, you'll want to consider: correlations between features and the target column, correlation with other features, and the variance of features.
- Along with correlation with other features, we need to also look for potential collinearity between some of the feature columns. Collinearity is when two feature columns are highly correlated and have the risk of duplicating information.
- We can generate a correlation matrix heatmap using Seaborn to visually compare the correlations and look for problematic pairwise feature correlations.
- Feature scaling helps ensure that some columns aren't weighted more than others when helping the model make predictions. We can rescale all of the columns to vary between 0 and 1. This is known as min-max scaling or rescaling. The formula for rescaling is as follows:

  $$\frac{x - \min(x)}{\max(x) - \min(x)}$$

  where x is the individual value, $\min(x)$ is the minimum value for the column x belongs to, and $\max(x)$ is the maximum value for the column x belongs to.

Resources

- seaborn.heatmap() documentation
- Feature scaling

Gradient Descent: Takeaways

Syntax

- Implementing gradient descent for 10 iterations:

```
a1_list = [1000]
alpha = 10
for x in range(0, 10):
    a1 = a1_list[x]
    deriv = derivative(a1, alpha, xi_list, yi_list)
    a1_new = a1 - alpha*deriv
    a1_list.append(a1_new)
```

Concepts

- The process of finding the optimal unique parameter values to form a unique linear regression model is known as model fitting. The goal of model fitting is to minimize the mean squared error between the predicted labels made using a given model and the true labels. The mean squared error is as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

- Gradient descent is an iterative technique for minimizing the squared error. Gradient descent works by trying different parameter values until the model with the lowest mean squared error is found. Gradient descent is a commonly used optimization technique for other models as well. An overview of the gradient descent algorithm is as follows:
    - Select initial values for the parameter a_1.
    - Repeat until convergence (usually implemented with a max number of iterations):
        - Calculate the error (MSE) of the model that uses current parameter value: $MSE(a_1) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2$
        - Calculate the derivative of the error (MSE) at the current parameter value: $\frac{d}{da_1}MSE(a_1)$
        - Update the parameter value by subtracting the derivative times a constant (α, called the learning rate): $a_1 := a_1 - \alpha\frac{d}{da_1}MSE(a_1)$
- Univariate case of gradient descent:
    - The function that we optimize through minimization is known as a cost function or as the loss function. In our case, the loss function is: $MSE(a_1) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2$
    - Applying calculus properties to simplify the derivative of the loss function:
        - Applying the linearity of differentiation property, we can bring the constant term outside the summation:
            - $\frac{d}{da_1}MSE(a_1) = \frac{1}{n}\sum_{i=1}^{n}\frac{d}{da_1}(a_1x_1^{(i)} - y^{(i)})^2$
        - Using the power rule and the chain rule to simplify:
            - $\frac{d}{da_1}MSE(a_1) = \frac{1}{n}\sum_{i=1}^{n}2(a_1x_1^{(i)} - y^{(i)})\frac{d}{da_1}(a_1x_1^{(i)} - y^{(i)})$
        - Because we're differentiating with respect to a_1, we treat $y^{(i)}$ and $x_1^{(i)}$ as constants.
            - $\frac{d}{da_1}MSE(a_1) = \frac{2}{n}\sum_{i=1}^{n}x_1^{(i)}(a_1x_1^{(i)} - y^{(i)})$

- For every iteration of gradient descent:
    - The derivative is computed using the current a_1 value.
    - The derivative is multiplied by the learning rate (α): $\alpha \frac{d}{da_1} MSE(a_1)$ The result is subtracted from the current parameter value and assigned as the new parameter value: $a_1 := a_1 - \alpha \frac{d}{da_1} MSE(a_1)$
- Multivariate case of gradient descent:
    - When we have two parameter values (a_0 and a_1), the cost function is now a function of two variables instead of one. Our new cost function is: $MSE(a_0, a_1) = \frac{1}{n} \sum_{i=1}^{n} (a_0 + a_1 x_1^{(i)} - y^{(i)})^2$
    - We also need two update rules:
        - $a_0 := a_0 - \alpha \frac{d}{da_0} MSE(a_0, a_1)$
        - $a_1 := a_1 - \alpha \frac{d}{da_1} MSE(a_0, a_1)$
    - Computed derivative for the multivariate case:
        - $\frac{d}{da_1} MSE(a_0, a_1) = \frac{2}{n} \sum_{i=1}^{n} x_1^{(i)} (a_0 + a_1 x_1^{(i)} - y^{(i)})$
- Gradient descent scales to as many variables as you want. Keep in mind each parameter value will need its own update rule, and it closely matches the update for a_1. The derivative for other parameters are also identical.
- Choosing good initial parameter values, and choosing a good learning rate are the main challenges with gradient descent.

Resources

- Mathematical Optimization
- Loss Function

Ordinary Least Squares: Takeaways

Syntax

- Finding the optimal parameter values using ordinary least squares (OLS):

```
first_term = np.linalg.inv(
    np.dot(
        np.transpose(X),
        X
    )
    )
second_term = np.dot(
    np.transpose(X),
    y
    )
a = np.dot(first_term, second_term)
print(a)
```

Concepts

- The ordinary least squares estimation provides a clear formula for directly calculating the optimal values that maximizes the cost function.
- The OLS estimation formula that results in optimal vector a:

$$a = (X^T X)^{-1} X^T y$$

- OLS estimation provides a closed form solution to the problem of finding the optimal parameter values. A closed form solution is where a solution can be computed arithmetically with a predictable amount of mathematical operations.
- The error for OLS estimation is often represented using the Greek letter for E. Since the error is the difference between the predictions made using the model and the actual labels, it's represented as a vector:

$$\epsilon = \hat{y} - y$$

- We can use the error metric to define y:

$$y = Xa - \epsilon$$

- The cost function in matrix form:

$$J(a) = \frac{1}{n}(Xa - y)^T (Xa - y)$$

- The derivative of the cost function:

$$\frac{dJ(a)}{da} = 2X^T Xa - 2X^T y$$

- Minimizing the cost function, $J(a)$.
    - Set the derivative equal to 0 and solve for a:
        - $2X^T Xa - 2X^T y = 0$
    - Compute the inverse of X and multiply both sides by the inverse:
        - $a = (X^T X)^{-1} X^T y$

- The biggest limitation of OLS estimation is that it's computationally expensive when the data is large. Computing the inverse of a matrix has a computational complexity of approximately $O(n^3)$.
- OLS is computationally expensive, and so is commonly used when the numbers of elements in the dataset is less than a few million elements.

Resources

- Walkthrough of the derivative of the cost function
- Ordinary least squares

Processing And Transforming Features: Takeaways

Syntax

- Converting any column to the categorical data type:

```
train['Utilities'] = train['Utilities'].astype('category')
```

- Accessing the underlying numerical representation of a column:

```
train['Utilities'].cat.codes
```

- Applying dummy coding for all of the text columns:

```
dummy_cols = pd.get_dummies()
```

- Replace all missing values in a column with a specified value:

```
fill_with_zero = missing_floats.fillna(0)
```

Concepts

- Feature engineering is the process of processing and creating new features. Feature engineering is a bit of an art and having knowledge in the specific domain can help create better features.
- Categorical features are features that can take on one of a limited number of possible values.
- A drawback to converting a column to the categorical data type is that one of the assumptions of linear regression is violated. Linear regression operates under the assumption that the features are linearly correlated with the target column.
- Instead of converting to the categorical data type, it's common to use a technique called dummy coding. In dummy coding, a dummy variable is used. A dummy variable that takes the value of 0 or 1 to indicate the absence or presence of some categorical effect that may be expected to shift the outcome.
- When values are missing in a column, there are two main approaches we can take:
    - Removing rows that contain missing values for specific columns:
        - Pro: Rows containing missing values are removed, leaving only clean data for modeling.
        - Con: Entire observations from the training set are removed, which can reduce overall prediction accuracy.
    - Imputing (or replacing) missing values using a descriptive statistic from the column:
        - Pro: Missing values are replaced with potentially similar estimates, preserving the rest of the observation in the model.
        - Con: Depending on the approach, we may be adding noisy data for the model to learn.

Resources

- Feature Engineering
- Dummy Coding

- pandas.Dataframe.ffillna()

Logistic regression: Takeaways

Syntax

- Defining the logistic function:

```
def logistic(x):
     """
     np.exp(x) raises x to the exponential power e^x. e ~= 2.71828
     """
return np.exp(x)  / (1 + np.exp(x))
```

- Instantiating a logistic regression model:

```
from sklearn.linear_model import LogisticRegression
linear_model = LogisticRegression()
```

- Training a logistic regression model:

```
logistic_model.fit(admissions[["gpa"]], admissions["admit"])
```

- Returning predicted probabilities for a column:

```
pred_probs = logistic_model.predict_proba(admission[["gpa"]])
```

Concepts

- In classification, our target column has a finite set of possible values, which represent different categories a row can belong to.
- In binary classification, there are only two options for values:
    - `0` for the False condition.
    - `1` for the True condition.
- Categorical values are used to represent different options or categories. Classification focuses on estimating the relationship between the independent variables and the dependent categorical variable.
- One technique of classification is called logistic regression. While a linear regression model outputs a real number as the label, a logistic regression model outputs a probability value.
- The logistic function is a version of the linear function that is adapted for classification. Mathematically, the logistic function is represented as the following:

$$\sigma(t) = \frac{e^t}{1+e^t}$$

where e^t is the exponential transformation to transform all values to be positive, and $\frac{t}{1+t}$ is the normalization transformation to transform all values between `0` and `1`.

Resources

- Documentation for the LogisticRegression class
- Documentation for the predict_proba method

Introduction to evaluating binary classifiers: Takeaways

Concepts

- Prediction accuracy is the simplest way to determine the effectiveness of a classification model. Prediction accuracy can be calculated by the number of labels correctly predicted divided the total number of observations:

$$Accuracy = \frac{\#\text{ of Correctly Predicted}}{\#\text{ of Observations}}$$

- A discrimination threshold is used to determine what labels are assigned based on their probability. Scikit-learn sets the discrimination threshold to 0.5 by default when predicting labels.
    - For example, if the predicted probability is greater than 0.5, the label for that observation is 1. If it is less than 0.5, the label for that observation is 0.
- There are four different outcomes of a binary classification model:
    - **True Positive:** The model correctly predicted the label as positive.
    - **True Negative:** The model correctly predicted the label as negative.
    - **False Positive:** The model falsely predicted the label as positive.
    - **False Negative:** The model falsely predicted the label as negative.
- Sensitivity or True Positive Rate, is the proportion of labels that were correctly predicted as positive. Mathematically, this is written as:

$$TPR = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

    - Sensitivity helps answer "How effective is this model at identifying positive outcomes?"
- Specificity or True Negative Rate, is the proportion of labels that were correctly predicted as negative. Mathematically, this is written as:

$$TNR = \frac{\text{True Negatives}}{\text{False Positives} + \text{True Negatives}}$$

    - Specificity helps answer "How effective is the model at identifying negative outcomes?"

Resources

- Sensitivity and Specificity
- Discrimination threshold

Multiclass classification: Takeaways

Syntax

- Returning a DataFrame containing binary columns:

```
dummy_df = pd.get_dummies(cars["cylinders"])
```

- Concatenating DataFrames:

```
cars = pd.concat([cars, dummy_df], axis=1)
```

- Returning a Series containing the index of the maximum value:

```
predicted_origins=testing_probs.idxmax(axis=1)
```

Concepts

- In the instance where two values are just two different labels, it is safer to turn the discrete values into categorical variables.
- Dummy variables are for columns that represent categorical values.
- A problem is a multiclass classification problem when there are three or more categories or classes. There are existing multiclassification techniques that can be categorized into the following:
    - Transformation to Binary: Reducing the problem to multiple binary classification problems.
    - Extension from Binary: Extending existing binary classifiers to solve multi-class classification problems.
    - Hierarchical classification: Dividing the output into a tree where each parent node is divided into multiple child nodes and the process is continued until each child node represents only one class.
- The one-versus-all method, which is a transformation to binary technique, is a technique where we choose a single category as the Positive case and group the rest of the categories as the False case.

Resources

- Documentation for idxmax()
- Multiclass Classification

Overfitting: Takeaways

Concepts

- Bias and variance are at the heart of understanding overfitting.
- Bias describes error that results in bad assumptions about the learning algorithm. Variance describes error that occurs because of the variability of a model's predicted values.
- We can approximate the bias of a model by training a few different models using different features on the same class and calculating their error scores.
- To detect overfitting, you can compare the in-sample error and the out-of-sample error, or the training error with the test error.
    - To calculate the out-of-sample error, you need to test the data on a test set of data. If you don't have a separate test data set, you can use cross-validation.
    - To calculate the in-sample-error, you can test the model over the same data it was trained on.
- When the out-of-sample error is much higher than the in-sample error, this is a clear indicator the trained model doesn't generalize well outside the training set.

Resources

- Bias-variance tradeoff
- Blog post on the bias-variance tradeoff

Clustering basics: Takeaways

Syntax

- Computing the Euclidean distance in Python:

```
from sklearn.metrics.pairwise import euclidean_distances
euclidean_distances(votes.iloc[0,3:], votes.iloc[1,3:])
```

- Initializing the KMeans class from scikit-learn:

```
from sklearn.cluster import KMeans
kmeans_model = KMeans(n_clusters=2, random_state=1)
```

- Calculating the distance between observations and the clusters:

```
senator_distances = kmeans_model.fit_transform(votes.iloc[:, 3:])
```

- Computing a frequency table of two or more factors:

```
labels = kmeans_model.labels_
print(pd.crosstab(labels, votes["party"]))
```

Concepts

- Two major types of machine learning are supervised and unsupervised learning. In supervised learning, you train an algorithm to predict an unknown variable from known variables. In unsupervised learning, you're finding patterns in data as opposed to making predictions.
- Unsupervised learning is very commonly used with large data sets where it isn't obvious how to start with supervised machine learning. It's a good idea to try unsupervised learning to explore a data set before trying to use supervised machine learning models.
- Clustering is one of the main unsupervised learning techniques. Clustering algorithms group similar rows together and is a key way to explore unknown data.
- We can use the Euclidean distance formula to find the distance between two rows to group similar rows. The formula for Euclidean distance is:

$$d = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \ldots + (q_n - p_n)^2}$$

where q_n and p_n are observations from each row.

- The k-means clustering algorithm uses Euclidean distance to form clusters of similar items.

Resources

- Documentation for sklearn.cluster.KMeans
- Unsupervised Machine learning
- Redefining NBA Basketball Positions

K-means clustering: Takeaways

Syntax

- Computing the Euclidean distance in Python:

```
def calculate_distance(vec1, vec2):
    root_distance = 0
    for x in range(0, len(vec1)):
        difference = centroid[x] - player_values[x]
        squared_difference = difference**2
        root_distance += squared_difference
euclid_distance = math.sqrt(root_distance)
return euclid_distance
```

- Assigning observations to clusters:

```
def assign_to_cluster(row):
lowest_distance = -1
closest_cluster = -1
for cluster_id, centroid in centroids_dict.items():
    df_row = [row['ppg'], row['atr']]
    euclidean_distance = calculate_distance(centroid, df_row)
    if lowest_distance == -1:
        lowest_distance = euclidean_distance
        closest_cluster = cluster_id
    elif euclidean_distance < lowest_distance:
        lowest_distance = euclidean_distance
        closest_cluster = cluster_id
return closest_cluster
```

- Initializing the KMeans class from scikit-learn:

```
from sklearn.cluster import KMeans
kmeans_model = KMeans(n_clusters=2, random_state=1)
```

Concepts

- Centroid-based clustering works well when the clusters resemble circles with centers.
- K-means clustering is a popular centroid-based clustering algorithm. The K refers to the number of clusters into which we want to segment our data. K-means clustering is an iterative algorithm that switches between recalculating the centroid of each cluster and the items that belong to each cluster.
- Euclidean distance is the most common measure of distance between vectors used in data science. Here is the formula for distance in two dimensions:
  $\sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \ldots + (q_n - p_n)^2}$
  where q and p are the two vectors we are comparing.
  - Example: If `q` is (5,2) and `p` is (3,1), the distance comes out to:
    $\sqrt{(5 - 3)^2 + (2 - 1)^2}$

$\sqrt{5}$
2.23607.

- If clusters look like they don't move much after every iteration, this means two things:
    - K-means clustering doesn't cause significant changes in the makeup of clusters between iterations, meaning that it will always converge and become stable.
    - Where we pick the initial centroids and how we assign elements to clusters initially matters a lot because K-means clustering is conservative between iterations.
- To counteract the problems listed above, the `sklearn` implementation of K-means clustering does some intelligent things like re-running the entire clustering process many times with random initial centroids so the final results are a little less biased.

Resources

- Sklearn implementation of K-Means clustering
- Implementing K-Means clustering from scratch

Introduction to Decision Trees: Takeaways

Syntax

- Converting a categorical variable to a numeric value:

```
col = pandas.Categorical(income["workclass"])
```

- Retrieving the numeric value of the categorical variable:

```
col.codes
```

- Using Python to calculate entropy:

```
def calc_entropy(column):
"""
Calculate entropy given a pandas series, list, or numpy array.
"""
counts = numpy.bincount(column)
probabilities = counts / len(column)
entropy = 0
for prob in probabilities:
    if prob > 0:
        entropy += prob * math.log(prob, 2)
return -entropy
```

- Using Python to calculate information gain:

```
def calc_information_gain(data, split_name, target_name):
"""
Calculate information gain given a data set, column to split on, and target
"""
original_entropy = calc_entropy(data[target_name])
column = data[split_name]
median = column.median()
left_split = data[column <= median]
right_split = data[column > median]
to_subtract = 0
for subset in [left_split, right_split]:
    prob = (subset.shape[0] / data.shape[0])
    to_subtract += prob * calc_entropy(subset[target_name])
return original_entropy - to_subtract
```

Concepts

- Decision trees are a powerful and popular machine learning technique. The decision machine learning algorithm enables us to automatically construct a decision tree that tells us what outcomes we should predict in certain situations.

- Decision trees can pick up nonlinear interactions between variables in the data that linear regression cannot.
- A decision tree is made up of a series of nodes and branches. A node is where we split the data based on a variable, and a branch is one side of the split. The tree accumulates more levels as the data is split based on variables.
- A tree is n levels deep where n is one more than the number of nodes. The nodes at the bottom of the tree are called terminal nodes, or leaves.
- When splitting the data, you aren't splitting randomly; there is an objective to make a prediction on future data. To meet complete our objective, each leaf must have only one value for our target column.
- One type of algorithm used to construct decision trees is called the ID3 algorithm. There are other algorithms like CART that use different metrics for the split criterion.
- A metric used to determine how "together" different values are is called entropy, which refers to disorder. For example, if there were many values "mixed together", the entropy value would be high while a dataset consisting of one value would have low entropy.
- The formula for entropy is $\sum_{i=1}^{c} P(x_i) \log_b P(x_i)$ where i is a unique value in a single column, $P(x_i)$ is the probability of the value occurring in our data, b is the base of our logarithm, and c is the number of unique values in a single column.
- You can use information gain to tell which split will reduce entropy the most.
- The formula for information gain is the following:
$$IG(T, A) = Entropy(T) - \sum_{v \in A} \frac{|T_v|}{|T|} \cdot Entropy(T_v)$$
IG is information gain, T is our target variable, A is the variable you are splitting on, and T_v is the number of times a unique value is the target variable.

Resources

- Pandas categorical class documentation
- Information Theory
- Entropy

Building a Decision Tree: Takeaways

Syntax

- Using Python to calculate entropy:

```
def calc_entropy(column):
"""
Calculate entropy given a pandas series, list, or numpy array.
"""
counts = numpy.bincount(column)
probabilities = counts / len(column)
entropy = 0
for prob in probabilities:
    if prob > 0:
        entropy += prob * math.log(prob, 2)
return -entropy
```

- Using Python to calculate information gain:

```
def calc_information_gain(data, split_name, target_name):
"""
Calculate information gain given a data set, column to split on, and target.
"""
original_entropy = calc_entropy(data[target_name])
column = data[split_name]
median = column.median()
left_split = data[column <= median]
right_split = data[column > median]
to_subtract = 0
for subset in [left_split, right_split]:
    prob = (subset.shape[0] / data.shape[0])
    to_subtract += prob * calc_entropy(subset[target_name])
return original_entropy - to_subtract
```

- Finding the best column to split on:

```
def find_best_column(data, target_name, columns):
"""
Find the best column to split on given a data set, target variable, and list of columns.
information_gains = []
for col in columns:
    information_gain = calc_information_gain(data, col, "high_income")
    information_gains.append(information_gain)
highest_gain_index = information_gains.index(max(information_gains))
highest_gain = columns[highest_gain_index]
return highest_gain
```

- Applying a function to a data frame:

```
df.apply(find_best_column, axis=0)
```

Concepts

- Pseudocode is a piece of plain-text outline of a piece of code explaining how the code works. Exploring the pseudocode is a good way to understand it before tying to code it.
- Pseudocode for the ID3 algorithm:

```
def id3(data, target, columns)
    1 Create a node for the tree
    2 If all values of the target attribute are 1, Return the node, with label = 1
    3 If all values of the target attribute are 0, Return the node, with label = 0
    4 Using information gain, find A, the column that splits the data best
    5 Find the median value in column A
    6 Split column A into values below or equal to the median (0), and values above the
median (1)
    7 For each possible value (0 or 1), vi, of A,
    8    Add a new tree branch below Root that corresponds to rows of data where A = vi
    9    Let Examples(vi) be the subset of examples that have the value vi for A
   10    Below this new branch add the subtree id3(data[A==vi], target, columns)
   11 Return Root
```

- We can store the entire tree in a nested dictionary by representing the root node with a dictionary and branches with keys for the left and right node.
- Dictionary for a decision tree:

```
{
   "left":{
  "left":{
     "left":{
        "number":4,
        "label":0
     },
     "column":"age",
     "median":22.5,
     "number":3,
     "right":{
        "number":5,
        "label":1
     }
  },
  "column":"age",
  "median":25.0,
  "number":2,
  "right":{
     "number":6,
     "label":1
  }
   },
```

```
   "column":"age",
   "median":37.5,
   "number":1,
   "right":{
  "left":{
     "left":{
        "number":9,
        "label":0
     },
     "column":"age",
     "median":47.5,
     "number":8,
     "right":{
        "number":10,
        "label":1
     }
  },
  "column":"age",
  "median":55.0,
  "number":7,
  "right":{
     "number":11,
     "label":0
  }
   }
}
```


Resources

- Recursion
- ID3 Algorithm

Applying Decision Trees: Takeaways

Syntax

- Instantiating the scikit-learn decision tree classifier:

```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(random_state=1)
```

- Fitting data using the decision tree classifier:

```
columns = ["age", "workclass", "education_num", "marital_status"]
clf.fit(income[columns], income["high_income"]
```

- Shuffling the rows of a data frame and re-indexing:

```
income = income.reindex(numpy.random.permutation(income.index))
```

- Calculating the area under curve (AUC) using scikit-learn:

```
from sklearn.metrics import roc_auc_score
error = roc_auc_score(test["high_income"], clf.predict(test[columns])
```

Concepts

- Scikit-learn includes the `DecisionTreeClassifier` class for classification problems, and `DecisionTreeRegressor` for regression problems.
- AUC (area under the curve) ranges from 0 to 1 and is a measure of how accurate our predictions are, which makes it ideal for binary classification. The higher the AUC, the more accurate our predictions. AUC takes in two parameters:
    - `y_true` : true labels.
    - `y_score` : predicted labels.
- Trees overfit when they have too much depth and make overly complex rules that match the training data but aren't able to generalize well to new data. The deeper the tree is, the worse it typically performs on new data.
- Three ways to combat overfitting:
    - "Prune" the tree after we build it to remove unnecessary leaves.
    - Use ensampling to blend the predictions of many trees.
    - Restrict the depth of the tree while we're building it.
- We can restrict tree depth by adding a few parameters when we initialize the `DecisionTreeClassifier` :
    - `max_depth` : Globally restricts how deep the tree can go.
    - `min_samples_split` : The minimum number of rows a node should have before it can be split; if this is set to 2 then nodes with two rows won't be split and will become leaves instead.
    - `min_samples_leaf` : The minimum number of rows a leaf must have.
    - `min_weight_fraction_leaf` : The fraction of input rows a leaf must have.

        - `max_leaf_nodes` : The maximum number of total leaves; this will limit the number of leaf nodes as the tree is being built.
        - However, some parameters aren't compatible. For example, we can't use `max_depth` and `max_leaf_nodes` together.
- Underfitting occurs when our model is too simple to explain the relationships between the variables.
- High bias can cause underfitting while high variance can cause overfitting. We call this bias-variance tradeoff because decreasing one characteristic will usually increase the other. This is a limitation of all machine learning algorithms.
- The main advantages of using decision trees is that decision trees are:
    - Easy to interpret.
    - Relatively fast to fit and make predictions.
    - Able to handle multiple types of data.
    - Able to pick up nonlinearities in data and fairly accurate.
- The most powerful way to reduce decision tree overfiting is to create ensembles of trees. The random forest algorithm is a popular choice for doing this. In cases where prediction accuracy is the most important consideration, random forests usually perform better.

Resources

- DecisionTreeClassifier class documentation
- Area under the curve
- Documentation for roc_auc_score

Introduction to Random Forests: Takeaways

Syntax

- Instantiating the RandomForestClassifier:

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(n_estimators=5, random_state=1, min_samples_leaf=2)
```

- Predicting the probability that a given class is correct for a row:

```
RandomForestClassifier.predict_proba()
```

- Computing the area under the curve:

```
print(roc_auc_score(test["high_income"], predictions))
```

- Introducing variation through bagging:

```
bag_proportion = .6
predictions = []
for i in range(tree_count):
    bag = train.sample(frac=bag_proportion, replace=True, random_state=i)
    clf = DecisionTreeClassifier(random_state=1, min_samples_leaf=2)
    clf.fit(bag[columns], bag["high_income"])
    predictions.append(clf.predict_proba(test[columns])[:,1])
combined = numpy.sum(predictions, axis=0) / 10
rounded = numpy.round(combined)
```

Concepts

- The random forest algorithm is a powerful tool to reduce overfitting in decision trees.
- Random forest is an ensemble algorithm that combines the predictions of multiple decision trees to create a more accurate final prediction.
- There are many methods to get from the output of multiple models to a final vector of predictions. One method is majority voting. In majority voting, each decision tree classifier gets a "vote" and the most commonly voted value for each row "wins."
- To get ensemble predictions, use the `predict_proba` method on the classifiers to generate probabilities, take the mean of the probabilities for each row, and then round the result.
- The more dissimilar the models we use to construct an ensemble are, the stronger their combined predictions will be. For example, ensembling a decision tree and a logistic regression model will result in stronger predictions than ensembling two decision trees with similar parameters. However, ensembling similar models will result in a negligible boost in the accuracy of the model.
- Variation in the random forest will ensure each decision tree is constructed slightly differently and will make different predictions as a result. Bagging and random forest subsets are two main ways to introduce variation in a random forest.
- With bagging, we train each tree on a random sample of the data or "bag". When doing this, we perform sampling with replacement, which means that each row may appear in the "bag"

multiple times. With random forest subsets, however, only a constrained set of features that is selected randomly will be used to introduce variation into the trees.

- `RandomForestClassifier` has an `n_estimators` parameter that allows you to indicate how many trees to build. While adding more trees usually improves accuracy, it also increases the overall time the model takes to train. The class also includes the `bootstrap` parameter which defaults to `True`. "Bootstrap aggregation" is another name for bagging.
- `RandomForestClassifier` has a similar interface to `DecisionTreeClassifier` and we can use the `fit()` and `predict()` methods to train and make predictions.
- The main strengths of a random forest are:
    - Very accurate predictions: Random forests achieve near state-of-the-art performance on many machine learning tasks.
    - Resistance to overfitting: Due to their construction, random forests are fairly resistant to overfitting.
- The main weaknesses of using a random forest are:
    - They're difficult to interpret: Since we've averaging the results of many trees, it can be hard to figure out why a random forest is making predictions the way it is.
    - They take longer to create: Making two trees takes twice as long as making one, making three trees takes three times as long, and so on.

Resources

- Majority Voting
- RandomForestClassifier documentation

Representing Neural Networks: Takeaways

Syntax

- Generating data with specific properties using scikit learn:
  - sklearn.datasets.make_regression()
  - sklearn.datasets.make_classification()
  - sklearn.datasets.make_moons()
- Generating a regression data set with 3 features, 1000 observations, and a random seed of 1:

```
from sklearn.datasets import make_regression
 data = make_regression(n_samples=1000, n_features=3, random_state=1)
```

- Returning a tuple of two NumPy objects that contain the generated data:

```
print(type(data))
 tuple
```

- Retrieving the features of the generated data:

```
print(data[0])
 array([[ 0.93514778,  1.81252782,  0.14010988],
       [-3.06414136,  0.11537031,  0.31742716],
       [-0.42914228,  1.20845633,  1.1157018 ],
       ...,
       [-0.42109689,  1.01057371,  0.20722995],
       [ 2.18697965,  0.44136444, -0.10015523],
       [ 0.440956  ,  0.32948997, -0.29257894]])
```

- Retrieving the first row of data:

```
print(data[0][0])
array([ 0.93514778,  1.81252782,  0.14010988])
```

- Retrieving the labels of the data:

```
print(data[1])
array([ 2.55521349e+02,  -2.24416730e+02,   1.77695808e+02,
       -1.78288470e+02,  -6.31736749e+01,  -5.52369226e+01,
       -2.33255554e+01,  -8.81410996e+01,  -1.75571964e+02,
       1.06048917e+01,   7.07568627e+01,   2.86371625e+02,
       ...,
       -7.38320267e+01,  -2.38437890e+02,  -1.23449719e+02,
        3.36130733e+01,  -2.67823475e+02,   1.21279169e+00,
       -2.62440408e+02,   1.32486453e+02,  -1.93414037e+02,
       -2.75702376e+01,  -1.00678877e+01,   2.05169507e+02,
       -1.52978767e+02,   1.18361239e+01,  -2.97505169e+02,
       2.40169605e+02,   7.33158364e+01,   2.18888903e+02,
       3.92751308e+01])
```

- Retrieving the first label of the data:

```
print(data[1][0])

255.52134901495128
```

- Creating a dataframe:

```
features = pd.DataFrame(data[0])
```

Concepts

- Neural networks are usually represented as **graphs**. A graph is a data structure that consists of nodes (represented as circles) that are connected by edges (represented as lines between the nodes).
- Graphs are a highly flexible data structure; you can even represent a list of values as a graph. Graphs are often categorized by their properties, which act as constraints. You can read about the many different ways graphs can be categorized on Wikipedia.
- Neural network models are represented as a **computational graph**. A computational graph uses nodes to describe variables and edges to describe how variables are combined.
- In a simple neural network:
    - each feature column in a data set is represented as an **input neuron**
    - each weight value is represented as an arrow from the feature column it multiples to the **output neuron**
- Inspired by biological neural networks, an **activation function** determines if the neuron *fires* or not. In a neural network model, the activation function transforms the weighted sum of the input values.

Resources

- Graph Theory on Wikipedia
- Directed Acyclic Graph on Wikipedia
- Feedforward Neural Network on Wikipedia
- Calculus on Computational Graphs
    - Explores how computational graphs can be used to organize derivatives.

Nonlinear Activation Functions: Takeaways

Syntax

- ReLU function:

```
def relu(values):
 return np.maximum(x, 0)
```

- Tan function:
    - np.tan
- Tanh function:
    - np.tanh

Concepts

TRIGONOMETRIC FUNCTIONS

- Trigonometry is short for triangle geometry and provides formulas, frameworks, and mental models for reasoning about triangles. Triangles are used extensively in theoretical and applied mathematics, and build on mathematical work done over many centuries.
- A triangle is a polygon that has the following properties:
    - 3 edges
    - 3 vertices
    - angles between edges add up to 180 degrees
- Two main ways that triangles can be classified is by the internal angles or by the edge lengths.
- A trigonometric function is a function that inputs an angle value (usually represented as θ) and outputs some value. These functions compute ratios between the edge lengths.
- The three main trigonometric functions:
    - $\sin(\theta) = \frac{opposite}{hypotenuse}$
    - $\cos(\theta) = \frac{adjacent}{hypotenuse}$
    - $\tan(\theta) = \frac{opposite}{adjacent}$
- Right triangle terminology:
    - Hypotenuse describes the line that isn't touching the right angle.
    - Opposite refers to the line opposite the angle.
    - Adjacent refers to the line touching the angle that *isn't* the hypotenuse.
- In a neural network model, we're able massively expand the expressivity of the model by adding one or more layers, each with multiple linear combinations and nonlinear transformations.
- The three most commonly used activation functions in neural networks are:
    - the sigmoid function
    - the ReLU function

- the tanh function

Resources

- Medium Post on Activation Functions
- Activation Function on Wikipedia
- Hyperbolic Tangent on Wikipedia
- Numpy documentation for tan
- Numpy documentation for tanh

Hidden Layers: Takeaways

Syntax

- Training a classification neural network:

```
from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(hidden_layer_sizes=(1,), activation='logistic')
```

- Training a regression neural network:

```
from sklearn.neural_network import MLPRegressor
mlp = MLPClassifier(hidden_layer_sizes=(1,), activation='relu')
```

- Specifying hidden layers as a tuple object:

```
mlp = MLPClassifier(hidden_layer_sizes=(1,), activation='relu')
mlp = MLPClassifier(hidden_layer_sizes=(10,10), activation='relu')
```

- Specifying the activation function:

```
mlp = MLPClassifier(hidden_layer_sizes=(1,), activation='relu')
mlp = MLPClassifier(hidden_layer_sizes=(10,10), activation='logistic')
mlp = MLPClassifier(hidden_layer_sizes=(10,10), activation='tanh')
```

- Generating data with nonlinearity:

```
from sklearn.datasets import make_moons
data = make_moons()
```

Concepts

- The intermediate layers are known as **hidden layers**, because they aren't directly represented in the input data or the output predictions. Instead, we can think of each hidden layer as intermediate features that are learned during the training process. This is actually very similar to how decision trees are structured. The branches and splits represent some intermediate features that are useful for making predictions and are analagous to the hidden layers in a neural network.
- Each of these hidden layers has its own set of weights and biases, which are discovered during the training process. In decision tree models, the intermediate features in the model represented something more concrete we can understand (feature ranges).
- The number of hidden layers and number of neurons in each hidden layer are hyperparameters that act as knobs for the model behavior.

Resources

- Sklearn Hyperparameter Optimization
- Neural Network Hyperparameter Optimization
- Deep Learning on Wikipedia