Data Science in der Versicherung - Vorbereitung für Vorstellungsgespräch

Sachversicherung | Operatives Controlling | Data Scientist

Inhaltsverzeichnis

1. Schadenfälle klassifizieren

1.1 Überblick

Die Klassifikation von Schadenfällen ist zentral für: - **Risikobewertung**: Identifikation von Mustern und Risikofaktoren - **Prämienkalkulation**: Anpassung von Tarifen basierend auf Schadenhäufigkeit - **Ressourcenplanung**: Optimale Zuteilung von Bearbeitungskapazitäten - **Betrugserkennung**: Identifikation verdächtiger Schadenfälle

1.2 Klassifikationsprobleme in der Versicherung

1.2.1 Binäre Klassifikation

- **Frage**: Ist ein Schadenfall legitim oder betrügerisch?
- **Ziel**: Automatische Erkennung von Betrugsfällen
- **Anwendung**: Erstbewertung neuer Schadenfälle

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix

# Beispiel: Betrugserkennung
# Features: Schadenshöhe, Zeitpunkt, Versicherungsdauer, Historie,
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size

model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

1.2.2 Multi-Klassen-Klassifikation

- **Frage**: Welcher Schadentyp liegt vor? (z.B. Brand, Wasserschaden, Diebstahl, Sturm)
- **Ziel**: Automatische Kategorisierung für besseres Routing

- **Anwendung**: Erstklassifikation bei Schadenmeldung

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, f1_score

# Multi-Klassen-Klassifikation für Schadentypen
model = LogisticRegression(multi_class='multinomial', solver='lbfgs
model.fit(X_train, y_train)

# Vorhersage und Evaluation
predictions = model.predict(X_test)
accuracy = accuracy_score(y_test, predictions)
f1_macro = f1_score(y_test, predictions, average='macro')
```

1.3 Wichtige Features für Schadenklassifikation

1.3.1 Versicherungsbezogene Features

- **Vertragsmerkmale**: Versicherungsdauer, Deckungsumfang, Selbstbeteiligung
- **Versicherungssumme**: Versicherter Wert
- **Tarifgruppe**: Bisherige Prämie, Risikoklasse
- **Historie**: Anzahl früherer Schäden, Schadenkosten in der Vergangenheit

1.3.2 Schadenbezogene Features

- **Schadenshöhe**: Geschätzter oder tatsächlicher Schaden
- **Schadentyp**: Kategorie (Brand, Wasser, etc.)
- **Schadenursache**: Beschreibung der Ursache
- **Zeitpunkt**: Datum, Saison, Tageszeit

1.3.3 Objektbezogene Features

- **Objekttyp**: Wohnung, Haus, Gewerbeobjekt
- **Standort**: PLZ, Region, Risikogebiet (Überschwemmung, etc.)
- **Alter/Größe**: Baujahr, Wohnfläche, etc.

- **Nutzung**: Wohnung, Büro, Lager, etc.

1.4 Praktische Implementierung

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import cross_val_score

# Datenaufbereitung
def prepare_schaden_data(df):
    """Schadensdaten für Klassifikation vorbereiten"""

    # Kategorische Variablen encodieren
    le = LabelEncoder()
    categorical_cols = ['schadentyp', 'objekttyp', 'region', 'ursac

    for col in categorical_cols:
        df[f'{col}_encoded'] = le.fit_transform(df[col].astype(str)

    # Numerische Features normalisieren
    scaler = StandardScaler()
    numerical_cols = ['schadenshoehe', 'versicherungssumme', 'versi
    df[numerical_cols] = scaler.fit_transform(df[numerical_cols])

    return df

# Modelltraining
def train_schaden_classifier(X, y):
    """Schadenklassifikator trainieren"""

    model = GradientBoostingClassifier(
        n_estimators=200,
        learning_rate=0.1,
        max_depth=5,
        random_state=42
    )

    # Cross-Validation
    cv_scores = cross_val_score(model, X, y, cv=5, scoring='f1_macr
    print(f"F1-Score (Cross-Validation): {cv_scores.mean():.3f} (+/
```

```
    # Finales Modell trainieren
    model.fit(X, y)
    return model

# Feature Importance analysieren
def analyze_feature_importance(model, feature_names):
    """Wichtigste Features für Klassifikation identifizieren"""

    importances = model.feature_importances_
    indices = np.argsort(importances)[::-1]

    print("\nTop 10 wichtigste Features:")
    for i in range(10):
        print(f"{i+1}. {feature_names[indices[i]]}: {importances[ir

    return importances
```

1.5 Anwendungsfälle im operativen Controlling

- **Automatisches Schaden-Routing**: Schadenfälle automatisch an richtige Abteilung weiterleiten
- **Ressourcenplanung**: Erwartete Bearbeitungszeiten schätzen
- **Betrugsprävention**: Verdächtige Fälle frühzeitig identifizieren
- **Kostensteuerung**: High-value Schäden priorisieren

---

2. Wahrscheinlichkeitsmodelle

2.1 Grundlagen

Wahrscheinlichkeitsmodelle sind Kern der Versicherungsmathematik und ermöglichen: - **Schadenwahrscheinlichkeit**: Wie wahrscheinlich ist ein Schaden? - **Schadenhöhe**: Wie hoch wird der Schaden erwartungsgemäß sein? - **Risikobewertung**: Kombination von Häufigkeit und Schwere

2.2 Häufigkeitsverteilungen (Claim Frequency)

2.2.1 Poisson-Verteilung

- **Anwendung**: Anzahl der Schäden pro Versicherungsperiode
- **Annahme**: Seltene, unabhängige Ereignisse
- **Parameter**: λ (Lambda) = erwartete Anzahl Schäden

```
from scipy import stats
import numpy as np

# Poisson-Verteilung für Schadenhäufigkeit
lambda_param = 0.5  # z.B. 0.5 Schäden pro Jahr im Durchschnitt

# Wahrscheinlichkeit für k Schäden
k = 2
prob_k_claims = stats.poisson.pmf(k, lambda_param)
print(f"Wahrscheinlichkeit für {k} Schäden: {prob_k_claims:.4f}")

# Kumulative Verteilung
prob_at_least_one = 1 - stats.poisson.pmf(0, lambda_param)
print(f"Wahrscheinlichkeit für mindestens 1 Schaden: {prob_at_least

# Beispiel: Häufigkeitsmodellierung für verschiedene Risikogruppen
risk_groups = {
    'niedrig': 0.1,   # 0.1 Schäden/Jahr
    'mittel': 0.5,    # 0.5 Schäden/Jahr
    'hoch': 1.2       # 1.2 Schäden/Jahr
}

for group, lambda_val in risk_groups.items():
    expected_claims = lambda_val
    print(f"{group.capitalize()}es Risiko: {expected_claims:.2f} er
```

2.2.2 Negative Binomialverteilung

- **Anwendung**: Wenn Varianz > Mittelwert (Überdispersion)

- **Vorteil**: Flexibler als Poisson, berücksichtigt Heterogenität
- **Parameter**: r (Anzahl Erfolge), p (Erfolgswahrscheinlichkeit)

```
# Negative Binomial für überdispergierte Schadenhäufigkeit
r = 2
p = 0.4

# Wahrscheinlichkeitsverteilung
k_values = np.arange(0, 6)
probs = stats.nbinom.pmf(k_values, r, p)

print("Negative Binomial - Schadenhäufigkeit:")
for k, prob in zip(k_values, probs):
    print(f"  {k} Schäden: {prob:.4f}")
```

2.3 Schwereverteilungen (Claim Severity)

2.3.1 Lognormale Verteilung

- **Anwendung**: Schadenhöhen (häufig für Sachversicherung)
- **Vorteil**: Positiv, rechtssteil (wie typische Schadenhöhen)
- **Parameter**: μ (Mittelwert des Logs), σ (Standardabweichung des Logs)

```
# Lognormale Verteilung für Schadenhöhen
mu = 7      # Mittelwert des Logarithmus
sigma = 1.5  # Standardabweichung des Logarithmus

# Zufällige Schadenhöhen generieren
sample_claims = stats.lognorm.rvs(s=sigma, scale=np.exp(mu), size=1

# Statistiken
mean_claim = np.mean(sample_claims)
median_claim = np.median(sample_claims)
print(f"Mittlere Schadenhöhe: €{mean_claim:,.2f}")
print(f"Median Schadenhöhe: €{median_claim:,.2f}")

# Wahrscheinlichkeit, dass Schaden > Schwellwert
threshold = 10000
prob_large = 1 - stats.lognorm.cdf(threshold, s=sigma, scale=np.exp
print(f"Wahrscheinlichkeit für Schaden > €{threshold:,.2f}: {prob_l
```

2.3.2 Gamma-Verteilung

- **Anwendung**: Alternative für Schadenhöhen
- **Vorteil**: Flexibel in Form, gut für verschiedene Schadentypen
- **Parameter**: α (Shape), β (Rate)

```
# Gamma-Verteilung für Schadenhöhen
alpha = 2
beta = 0.001  # Rate parameter

# Erwartungswert und Varianz
expected_value = alpha / beta
variance = alpha / (beta ** 2)
print(f"Erwartete Schadenhöhe: €{expected_value:,.2f}")
print(f"Varianz: €{variance:,.2f}")
```

2.3.3 Pareto-Verteilung

- **Anwendung**: Große Schäden (Tail-Risiko)

- **Vorteil**: Modelliert seltene, aber sehr hohe Schäden
- **Wichtig**: Für Extremwertanalyse

```
# Pareto-Verteilung für extreme Schäden
alpha = 2.5  # Shape parameter
xmin = 10000  # Minimum-Schwellwert

# Wahrscheinlichkeit für sehr hohe Schäden
very_high_claim = 50000
prob_extreme = 1 - stats.pareto.cdf(very_high_claim / xmin, alpha)
print(f"Wahrscheinlichkeit für Schaden > €{very_high_claim:,.2f}: {
```

2.4 Compound-Modelle (Häufigkeit × Schwere)

2.4.1 Gesamtschadenverteilung

Gesamtschaden = Summe aller einzelnen Schäden in einer Periode

```
def simulate_total_loss(frequency_param, severity_params, n_simulat
    """
    Simuliere Gesamtschadenverteilung

    Parameters:
    - frequency_param: Lambda für Poisson (Häufigkeit)
    - severity_params: (mu, sigma) für Lognormal (Schwere)
    - n_simulations: Anzahl Simulationen
    """
    total_losses = []

    for _ in range(n_simulations):
        # Anzahl Schäden
        n_claims = stats.poisson.rvs(frequency_param)

        # Summe der Schadenhöhen
        if n_claims > 0:
            individual_claims = stats.lognorm.rvs(
                s=severity_params[1],
                scale=np.exp(severity_params[0]),
                size=n_claims
            )
            total_loss = np.sum(individual_claims)
        else:
            total_loss = 0

        total_losses.append(total_loss)

    return np.array(total_losses)

# Beispiel: Gesamtschaden pro Jahr simulieren
lambda_annual = 0.8  # 0.8 Schäden pro Jahr im Durchschnitt
mu_severity, sigma_severity = 7, 1.5  # Lognormal Parameter für Sch

total_losses = simulate_total_loss(lambda_annual, (mu_severity, sig

# Statistiken
mean_total = np.mean(total_losses)
var95 = np.percentile(total_losses, 95)
var99 = np.percentile(total_losses, 99)
```

```
print(f"Erwarteter Gesamtschaden: €{mean_total:,.2f}")
print(f"Value at Risk (95%): €{var95:,.2f}")
print(f"Value at Risk (99%): €{var99:,.2f}")
```

2.5 Regressionsmodelle für Wahrscheinlichkeiten

2.5.1 Poisson-Regression

- **Ziel**: Schadenhäufigkeit basierend auf Risikofaktoren modellieren
- **Anwendung**: GLM (Generalized Linear Model)

```
from sklearn.linear_model import PoissonRegressor
import statsmodels.api as sm
from statsmodels.discrete.discrete_model import Poisson

# Poisson Regression für Häufigkeitsmodellierung
# Beispiel: Schadenhäufigkeit basierend auf Objekttyp, Region, etc.

# Mit statsmodels (bessere Statistiken)
X = sm.add_constant(X_features)  # Features mit Konstante
y_frequency = df['anzahl_schaeden']  # Anzahl Schäden als Zielvaria

poisson_model = Poisson(y_frequency, X)
poisson_results = poisson_model.fit()

print(poisson_results.summary())

# Mit scikit-learn (einfacher zu nutzen)
model = PoissonRegressor(alpha=0.1)
model.fit(X_features, y_frequency)

# Vorhersage: Erwartete Anzahl Schäden für neuen Vertrag
expected_frequency = model.predict(new_contract_features)
```

2.5.2 Gamma-Regression

- **Ziel**: Schadenhöhe basierend auf Risikofaktoren modellieren

- **Link-Funktion**: Logarithmus (log-link)

```
from statsmodels.genmod.generalized_linear_model import GLM
from statsmodels.genmod import families

# Gamma Regression für Schadenhöhenmodellierung
y_severity = df['schadenshoehe']  # Schadenhöhe als Zielvariable
X_features = df[['objekttyp', 'region', 'alter', 'versicherungssumm

# GLM mit Gamma-Familie und log-link
gamma_model = GLM(y_severity, X_features, family=families.Gamma(fam
gamma_results = gamma_model.fit()

print(gamma_results.summary())
```

2.6 Anwendungsfälle im operativen Controlling

- **Reservierung**: Statistische Reserven für offene Schäden
- **Prämienkalkulation**: Erwartete Schadenkosten pro Risiko
- **Risikomanagement**: VaR (Value at Risk) Berechnung
- **Planung**: Prognose erwarteter Schadenkosten

---

3. Tarifierungsmerkmale im Vertragsbereich

3.1 Überblick

Tarifierungsmerkmale sind Variablen, die zur Prämienberechnung verwendet werden: - **Ziel**: Risiko-adäquate Prämien - **Prinzip**: Gleiche Risiken → gleiche Prämien - **Rechtliche Anforderungen**: Transparenz, Diskriminierungsverbot

3.2 Typische Tarifierungsmerkmale in der Sachversicherung

3.2.1 Objektbezogene Merkmale

- **Objekttyp**: Wohnung, Haus, Gewerbeobjekt
- **Baujahr**: Alter des Gebäudes
- **Gebäudeart**: Massivbau, Holzbau, etc.
- **Größe**: Wohnfläche, Grundstücksgröße
- **Wohnsituation**: Eigennutzung, Vermietung

3.2.2 Standortmerkmale

- **Region/PLZ**: Geografische Lage
- **Risikogebiet**: Überschwemmungsgebiet, Erdbebenzone
- **Kriminalitätsrate**: Einbruchrisiko
- **Infrastruktur**: Feuerwehr-Erreichbarkeit

3.2.3 Vertragsbezogene Merkmale

- **Versicherungssumme**: Höhe der Deckung
- **Selbstbeteiligung**: Eigenanteil bei Schaden
- **Deckungsumfang**: Welche Risiken sind abgedeckt
- **Laufzeit**: Vertragsdauer

3.2.4 Historische Merkmale

- **Schadenhistorie**: Anzahl/Schwere früherer Schäden
- **Vertragsdauer**: Seit wann bei dieser Versicherung
- **Wechselhäufigkeit**: Versichererwechsel in der Vergangenheit

3.3 Feature Engineering für Tarifierung

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.feature_selection import mutual_info_regression

def prepare_tarification_features(df):
    """Tarifierungsmerkmale aufbereiten"""

    # 1. Kategorische Variablen encodieren
    categorical_cols = ['objekttyp', 'region', 'bauart', 'deckungsu

    # Option 1: One-Hot Encoding
    df_encoded = pd.get_dummies(df, columns=categorical_cols, prefi

    # Option 2: Target Encoding (besser für stark kardinale Variabl
    for col in categorical_cols:
        target_mean = df.groupby(col)['praemie'].mean()
        df[f'{col}_target_encoded'] = df[col].map(target_mean)

    # 2. Interaktionsterme erstellen
    df['wohnflaeche_x_objekttyp'] = df['wohnflaeche'] * df['objektt
    df['alter_x_region'] = df['baujahr'] * df['region_encoded']

    # 3. Transformationen
    df['log_versicherungssumme'] = np.log1p(df['versicherungssumme'
    df['versicherungssumme_per_qm'] = df['versicherungssumme'] / df

    # 4. Zeitbasierte Features
    df['vertragsdauer_jahre'] = (pd.Timestamp.now() - df['vertragsb
    df['saison'] = df['vertragsbeginn'].dt.month.map({
        12: 'Winter', 1: 'Winter', 2: 'Winter',
        3: 'Frühling', 4: 'Frühling', 5: 'Frühling',
        6: 'Sommer', 7: 'Sommer', 8: 'Sommer',
        9: 'Herbst', 10: 'Herbst', 11: 'Herbst'
    })

    return df
```

```
# Feature Importance für Tarifierung
def analyze_tarification_features(X, y):
    """Wichtige Tarifierungsmerkmale identifizieren"""

    # Mutual Information Score
    mi_scores = mutual_info_regression(X, y, random_state=42)

    # Feature-Wichtigkeit sortieren
    feature_importance = pd.DataFrame({
        'feature': X.columns,
        'importance': mi_scores
    }).sort_values('importance', ascending=False)

    print("Top 10 wichtigste Tarifierungsmerkmale:")
    print(feature_importance.head(10))

    return feature_importance
```

3.4 Prämienmodellierung

3.4.1 Generalisiertes Lineares Modell (GLM)

GLM ist Standard in der Versicherungstarifierung.

```
import statsmodels.api as sm
from statsmodels.genmod import families, links

def build_glm_premium_model(df):
    """GLM-Modell für Prämienkalkulation"""

    # Zielvariable: Prämie (oder Schadenkosten)
    y = df['praemie']

    # Features
    X = df[['objekttyp', 'region', 'wohnflaeche', 'versicherungssum
            'baujahr', 'selbstbeteiligung', 'vertragsdauer']]

    # Dummies erstellen
    X_encoded = pd.get_dummies(X, drop_first=True)
    X_encoded = sm.add_constant(X_encoded)

    # GLM mit Gamma-Familie (für Prämien/Schadenkosten)
    # Log-Link für multiplikative Effekte
    glm_model = sm.GLM(
        y,
        X_encoded,
        family=families.Gamma(link=links.log())
    )

    results = glm_model.fit()

    print(results.summary())

    # Prämienvorhersage für neuen Vertrag
    def predict_premium(new_contract):
        """Prämie für neuen Vertrag vorhersagen"""
        new_X = prepare_features(new_contract)
        premium = results.predict(new_X)
        return premium

    return results, predict_premium
```

```
# Alternative: GAM (Generalized Additive Model)
# Für nicht-lineare Zusammenhänge
```

3.4.2 XGBoost für Tarifierung

Moderne ML-Ansätze können GLM ergänzen.

```
import xgboost as xgb
from sklearn.model_selection import train_test_split

def build_xgboost_premium_model(df):
    """XGBoost-Modell für Prämienkalkulation"""

    # Features und Zielvariable
    feature_cols = ['objekttyp', 'region', 'wohnflaeche', 'versiche
                    'baujahr', 'selbstbeteiligung', 'schadenhistori

    X = pd.get_dummies(df[feature_cols])
    y = df['praemie']

    # Train/Test Split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_

    # XGBoost Modell
    model = xgb.XGBRegressor(
        n_estimators=200,
        learning_rate=0.1,
        max_depth=5,
        objective='reg:squarederror'
    )

    model.fit(X_train, y_train)

    # Evaluation
    train_pred = model.predict(X_train)
    test_pred = model.predict(X_test)

    from sklearn.metrics import mean_absolute_error, r2_score

    train_mae = mean_absolute_error(y_train, train_pred)
    test_mae = mean_absolute_error(y_test, test_pred)
    r2 = r2_score(y_test, test_pred)

    print(f"Train MAE: €{train_mae:,.2f}")
    print(f"Test MAE: €{test_mae:,.2f}")
    print(f"R² Score: {r2:.4f}")
```

```
    return model
```

3.5 Tarifstrukturanalyse

```
def analyze_tariff_structure(df):
    """Tarifstruktur analysieren"""

    # Prämie nach Risikogruppen
    tariff_analysis = df.groupby(['objekttyp', 'region']).agg({
        'praemie': ['mean', 'median', 'std', 'count'],
        'schadenshoehe': 'mean',
        'anzahl_schaeden': 'mean'
    }).round(2)

    # Loss Ratio nach Tarifgruppe
    df['loss_ratio'] = df['schadenshoehe'] / df['praemie']

    loss_ratio_by_tariff = df.groupby(['objekttyp', 'region']).agg(
        'loss_ratio': ['mean', 'count'],
        'praemie': 'sum',
        'schadenshoehe': 'sum'
    })

    print("Loss Ratio nach Tarifgruppe:")
    print(loss_ratio_by_tariff)

    # Tarifgruppen mit schlechter Performance identifizieren
    problem_tariffs = loss_ratio_by_tariff[
        loss_ratio_by_tariff['loss_ratio']['mean'] > 1.0
    ]

    print("\nTarifgruppen mit Loss Ratio > 100%:")
    print(problem_tariffs)

    return tariff_analysis, loss_ratio_by_tariff
```

3.6 Anwendungsfälle im operativen Controlling

- **Tarifanpassung**: Identifikation von Unter-/Übertarifierung
- **Performance-Monitoring**: Loss Ratio nach Tarifgruppen
- **Marktpositionierung**: Wettbewerbsanalyse der Prämien
- **Risikosegmentierung**: Differenzierung nach Risikogruppen

---

4. Kennzahlensystem von Schadensdaten

4.1 Überblick

Ein Kennzahlensystem ermöglicht: - **Operatives Controlling**: Tägliches Monitoring der Schadenperformance - **Früherkennung**: Trends und Anomalien schnell erkennen - **Steuerung**: Entscheidungsgrundlage für Management - **Reporting**: Strukturierte Berichterstattung

4.2 Zentrale Schaden-Kennzahlen

4.2.1 Häufigkeitskennzahlen

Loss Frequency Rate (LFR)

Formel: `Anzahl Schäden / Anzahl Versicherungsjahre`

```
def calculate_loss_frequency_rate(df):
    """Loss Frequency Rate berechnen"""

    total_claims = df['anzahl_schaeden'].sum()
    total_exposure = df['anzahl_versicherungsjahre'].sum()

    lfr = total_claims / total_exposure
    print(f"Loss Frequency Rate: {lfr:.4f} Schäden pro Versicherung

    # LFR nach Segmenten
    lfr_by_segment = df.groupby('segment').apply(
        lambda x: x['anzahl_schaeden'].sum() / x['anzahl_versicheru
    )

    return lfr, lfr_by_segment
```

Claim Frequency

Formel: `Anzahl Schäden / Anzahl Verträge`

```
def calculate_claim_frequency(df):
    """Claim Frequency berechnen"""

    total_claims = df['anzahl_schaeden'].sum()
    total_policies = len(df)

    claim_freq = total_claims / total_policies
    print(f"Claim Frequency: {claim_freq:.4f} Schäden pro Vertrag")

    return claim_freq
```

4.2.2 Schwerekennzahlen

Average Loss per Claim (ALC)

Formel: `Gesamtschadenhöhe / Anzahl Schäden`

```
def calculate_average_loss_per_claim(df):
    """Durchschnittliche Schadenhöhe pro Schadenfall"""

    total_loss = df['schadenshoehe'].sum()
    total_claims = df['anzahl_schaeden'].sum()

    alc = total_loss / total_claims
    print(f"Average Loss per Claim: €{alc:,.2f}")

    # ALC nach Schadentyp
    alc_by_type = df.groupby('schadentyp').apply(
        lambda x: x['schadenshoehe'].sum() / x['anzahl_schaeden'].s
    )

    return alc, alc_by_type
```

Average Loss per Policy (ALP)

Formel: `Gesamtschadenhöhe / Anzahl Verträge`

```
def calculate_average_loss_per_policy(df):
    """Durchschnittliche Schadenhöhe pro Vertrag"""

    total_loss = df['schadenshoehe'].sum()
    total_policies = len(df)

    alp = total_loss / total_policies
    print(f"Average Loss per Policy: €{alp:,.2f}")

    return alp
```

Median Loss

Wichtig für robuste Kennzahl (weniger anfällig für Ausreißer).

```
def calculate_median_loss(df):
    """Median-Schadenhöhe"""

    median_loss = df['schadenshoehe'].median()
    print(f"Median Loss: €{median_loss:,.2f}")

    # Vergleich Median vs. Mittelwert
    mean_loss = df['schadenshoehe'].mean()
    print(f"Mean Loss: €{mean_loss:,.2f}")
    print(f"Differenz: €{mean_loss - median_loss:,.2f}")

    return median_loss
```

4.2.3 Kombinierte Kennzahlen

Loss Ratio (Schadenquote)

Formel: `(Gesamtschadenhöhe / Gesamtprämien) × 100%`

```
def calculate_loss_ratio(df):
    """Loss Ratio berechnen"""

    total_loss = df['schadenshoehe'].sum()
    total_premium = df['praemie'].sum()

    loss_ratio = (total_loss / total_premium) * 100
    print(f"Loss Ratio: {loss_ratio:.2f}%")

    # Loss Ratio Ziel: < 100% (Versicherung macht Gewinn)
    if loss_ratio > 100:
        print("⚠️ Warnung: Loss Ratio über 100% - Verlust!")

    # Loss Ratio nach Segmenten
    loss_ratio_by_segment = df.groupby('segment').apply(
        lambda x: (x['schadenshoehe'].sum() / x['praemie'].sum()) *
    )

    return loss_ratio, loss_ratio_by_segment
```

Combined Ratio

Formel: `Loss Ratio + Expense Ratio`

```
def calculate_combined_ratio(df, total_expenses):
    """Combined Ratio berechnen"""

    total_loss = df['schadenshoehe'].sum()
    total_premium = df['praemie'].sum()

    loss_ratio = (total_loss / total_premium) * 100
    expense_ratio = (total_expenses / total_premium) * 100

    combined_ratio = loss_ratio + expense_ratio
    print(f"Loss Ratio: {loss_ratio:.2f}%")
    print(f"Expense Ratio: {expense_ratio:.2f}%")
    print(f"Combined Ratio: {combined_ratio:.2f}%")

    return combined_ratio
```

4.2.4 Weitere wichtige Kennzahlen

Claim Severity Index

Formel: `(Aktuelle ALC / Historische ALC) × 100`

```
def calculate_claim_severity_index(current_alc, historical_alc):
    """Claim Severity Index - Trend in Schadenhöhe"""

    severity_index = (current_alc / historical_alc) * 100
    print(f"Claim Severity Index: {severity_index:.2f}")

    if severity_index > 100:
        print("⚠️ Schadenhöhen steigen im Vergleich zur Historie")

    return severity_index
```

IBNR (Incurred But Not Reported)

Geschätzte Schäden, die noch nicht gemeldet wurden.

```
def estimate_ibnr(df, development_factor):
    """
    IBNR-Schätzung
    development_factor: Faktor basierend auf historischen Daten
    """

    reported_claims = df['gemeldete_schaeden'].sum()
    ibnr = reported_claims * development_factor

    print(f"Gemeldete Schäden: €{reported_claims:,.2f}")
    print(f"Geschätztes IBNR: €{ibnr:,.2f}")
    print(f"Gesamtschaden (inkl. IBNR): €{reported_claims + ibnr:,.

    return ibnr
```

4.3 Dashboard-Kennzahlen

```
import pandas as pd
import numpy as np
from datetime import datetime, timedelta

def create_schaden_dashboard(df, reporting_date):
    """Vollständiges Schaden-Dashboard erstellen"""

    dashboard = {}

    # 1. Übersichtskennzahlen
    dashboard['total_claims'] = df['anzahl_schaeden'].sum()
    dashboard['total_loss'] = df['schadenshoehe'].sum()
    dashboard['total_premium'] = df['praemie'].sum()
    dashboard['loss_ratio'] = (dashboard['total_loss'] / dashboard[

    # 2. Häufigkeitskennzahlen
    total_exposure = df['anzahl_versicherungsjahre'].sum()
    dashboard['loss_frequency_rate'] = dashboard['total_claims'] /

    # 3. Schwerekennzahlen
    dashboard['average_loss_per_claim'] = dashboard['total_loss'] /
    dashboard['median_loss'] = df['schadenshoehe'].median()

    # 4. Trends (vs. Vormonat/Vorjahr)
    # Hier würde man historische Daten vergleichen

    # 5. Segment-Analyse
    segment_analysis = df.groupby('segment').agg({
        'schadenshoehe': 'sum',
        'praemie': 'sum',
        'anzahl_schaeden': 'sum'
    })
    segment_analysis['loss_ratio'] = (
        segment_analysis['schadenshoehe'] / segment_analysis['praem
    ) * 100

    dashboard['segment_analysis'] = segment_analysis
```

```
    return dashboard

# Visualisierung der Kennzahlen
def visualize_kpis(df):
    """KPIs visualisieren"""

    import matplotlib.pyplot as plt
    import seaborn as sns

    fig, axes = plt.subplots(2, 2, figsize=(15, 12))

    # Loss Ratio Trend
    df_monthly = df.groupby(df['datum'].dt.to_period('M')).agg({
        'schadenshoehe': 'sum',
        'praemie': 'sum'
    })
    df_monthly['loss_ratio'] = (df_monthly['schadenshoehe'] / df_mo

    axes[0, 0].plot(df_monthly.index.astype(str), df_monthly['loss_
    axes[0, 0].axhline(y=100, color='r', linestyle='--', label='Bre
    axes[0, 0].set_title('Loss Ratio Trend')
    axes[0, 0].set_ylabel('Loss Ratio (%)')
    axes[0, 0].legend()

    # Schadentyp-Verteilung
    claim_type_dist = df.groupby('schadentyp')['schadenshoehe'].sum
    axes[0, 1].pie(claim_type_dist, labels=claim_type_dist.index, a
    axes[0, 1].set_title('Schadentyp-Verteilung (nach Volumen)')

    # Schadenhöhen-Verteilung
    axes[1, 0].hist(df['schadenshoehe'], bins=50, edgecolor='black'
    axes[1, 0].set_title('Verteilung der Schadenhöhen')
    axes[1, 0].set_xlabel('Schadenhöhe (€)')
    axes[1, 0].set_ylabel('Häufigkeit')

    # Loss Ratio nach Segmenten
    segment_lr = df.groupby('segment').apply(
        lambda x: (x['schadenshoehe'].sum() / x['praemie'].sum()) *
    )
    axes[1, 1].barh(segment_lr.index, segment_lr.values)
    axes[1, 1].axvline(x=100, color='r', linestyle='--')
```

```
    axes[1, 1].set_title('Loss Ratio nach Segmenten')
    axes[1, 1].set_xlabel('Loss Ratio (%)')

    plt.tight_layout()
    plt.show()
```

4.4 Anwendungsfälle im operativen Controlling

- **Tägliches Monitoring**: Dashboards mit aktuellen Kennzahlen
- **Früherkennung**: Alerts bei Abweichungen
- **Performance-Tracking**: Vergleich von Soll/Ist
- **Reporting**: Monatliche/Quartalsberichte für Management

---

5. Messwerte zur Modellgüteprüfung

5.1 Überblick

Modellgüteprüfung ist essentiell für: - **Vertrauen in Modellergebnisse**: Sind Vorhersagen zuverlässig? - **Modellauswahl**: Welches Modell ist am besten? - **Calibration**: Stimmen Wahrscheinlichkeitsvorhersagen? - **Bias-Erkennung**: Gibt es systematische Fehler?

5.2 Metriken für Klassifikation

5.2.1 Confusion Matrix

Basis für viele Metriken.

```
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt

def evaluate_classification_model(y_true, y_pred, labels=None):
    """Klassifikationsmodell evaluieren"""

    # Confusion Matrix
    cm = confusion_matrix(y_true, y_pred, labels=labels)

    # Visualisierung
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=labels, yticklabels=labels)
    plt.ylabel('Tatsächliche Klasse')
    plt.xlabel('Vorhergesagte Klasse')
    plt.title('Confusion Matrix')
    plt.show()

    # Detaillierter Report
    print(classification_report(y_true, y_pred, labels=labels))

    return cm
```

5.2.2 Accuracy (Genauigkeit)

Formel: `(TP + TN) / (TP + TN + FP + FN)`

```
from sklearn.metrics import accuracy_score

def calculate_accuracy(y_true, y_pred):
    """Accuracy berechnen"""

    accuracy = accuracy_score(y_true, y_pred)
    print(f"Accuracy: {accuracy:.4f} ({accuracy*100:.2f}%)")

    # ⚠️ Vorsicht bei unbalancierten Klassen!
    # Bei unbalancierten Daten ist Accuracy oft irreführend

    return accuracy
```

5.2.3 Precision (Präzision)

Formel: `TP / (TP + FP)`

Frage: Von den als positiv klassifizierten Fällen, wie viele sind tatsächlich positiv?

```
from sklearn.metrics import precision_score

def calculate_precision(y_true, y_pred, average='binary'):
    """Precision berechnen"""

    precision = precision_score(y_true, y_pred, average=average)

    if average == 'binary':
        print(f"Precision: {precision:.4f}")
    else:
        print(f"Precision ({average}): {precision:.4f}")

    # Beispiel: Betrugserkennung
    # Hohe Precision = Wenige False Positives
    # Wichtig: Bearbeiter sollen nicht viele Falschmeldungen bekomm

    return precision
```

5.2.4 Recall (Sensitivität, Trefferquote)

Formel: `TP / (TP + FN)`

Frage: Von allen tatsächlich positiven Fällen, wie viele haben wir gefunden?

```
from sklearn.metrics import recall_score

def calculate_recall(y_true, y_pred, average='binary'):
    """Recall berechnen"""

    recall = recall_score(y_true, y_pred, average=average)

    if average == 'binary':
        print(f"Recall: {recall:.4f}")
    else:
        print(f"Recall ({average}): {recall:.4f}")

    # Beispiel: Betrugserkennung
    # Hoher Recall = Wenige False Negatives
    # Wichtig: Möglichst viele Betrugsfälle sollen erkannt werden

    return recall
```

5.2.5 F1-Score

Formel: `2 × (Precision × Recall) / (Precision + Recall)`

Harmonisches Mittel aus Precision und Recall.

```
from sklearn.metrics import f1_score

def calculate_f1_score(y_true, y_pred, average='macro'):
    """F1-Score berechnen"""

    f1 = f1_score(y_true, y_pred, average=average)

    print(f"F1-Score ({average}): {f1:.4f}")

    # F1-Score ist gut, wenn Precision und Recall beide wichtig sin
    # Macro: Durchschnitt über alle Klassen (gilt für unbalancierte
    # Weighted: Gewichtet nach Klassen-Häufigkeit

    return f1
```

5.2.6 ROC-AUC (Receiver Operating Characteristic - Area Under Curve)

Misst die Fähigkeit, zwischen Klassen zu unterscheiden.

```
from sklearn.metrics import roc_auc_score, roc_curve
import matplotlib.pyplot as plt

def evaluate_roc_auc(y_true, y_pred_proba):
    """ROC-AUC evaluieren"""

    auc_score = roc_auc_score(y_true, y_pred_proba)
    print(f"ROC-AUC Score: {auc_score:.4f}")

    # Interpretation:
    # 0.5 = Zufällig (schlechtestes Modell)
    # 1.0 = Perfekt (bestes Modell)
    # > 0.7 = Gute Diskriminationsfähigkeit
    # > 0.8 = Sehr gute Diskriminationsfähigkeit

    # ROC Curve visualisieren
    fpr, tpr, thresholds = roc_curve(y_true, y_pred_proba)

    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, label=f'ROC Curve (AUC = {auc_score:.4f})')
    plt.plot([0, 1], [0, 1], 'k--', label='Random Classifier')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.legend()
    plt.grid(True)
    plt.show()

    return auc_score, fpr, tpr
```

5.2.7 Precision-Recall Curve

Besonders wichtig bei unbalancierten Klassen.

```
from sklearn.metrics import precision_recall_curve, average_precisi

def evaluate_precision_recall(y_true, y_pred_proba):
    """Precision-Recall Curve evaluieren"""

    precision, recall, thresholds = precision_recall_curve(y_true,
    avg_precision = average_precision_score(y_true, y_pred_proba)

    print(f"Average Precision: {avg_precision:.4f}")

    # Visualisierung
    plt.figure(figsize=(8, 6))
    plt.plot(recall, precision, label=f'PR Curve (AP = {avg_precisi
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title('Precision-Recall Curve')
    plt.legend()
    plt.grid(True)
    plt.show()

    return precision, recall, avg_precision
```

5.2.8 Brier Score

Für Wahrscheinlichkeitsvorhersagen (Calibration).

```
from sklearn.metrics import brier_score_loss

def calculate_brier_score(y_true, y_pred_proba):
    """Brier Score berechnen (niedriger ist besser)"""

    brier = brier_score_loss(y_true, y_pred_proba)
    print(f"Brier Score: {brier:.4f}")

    # Interpretation:
    # 0.0 = Perfekte Vorhersage
    # 0.25 = Zufällige Vorhersage (für binäres Problem)
    # Niedriger = Besser

    return brier
```

5.3 Metriken für Regression

5.3.1 MAE (Mean Absolute Error)

Formel: `mean(|y_true - y_pred|)`

```
from sklearn.metrics import mean_absolute_error

def calculate_mae(y_true, y_pred):
    """Mean Absolute Error berechnen"""

    mae = mean_absolute_error(y_true, y_pred)
    print(f"MAE: €{mae:,.2f}")

    # Interpretation: Durchschnittlicher Fehler in absoluten Werter
    # Beispiel: MAE von €500 bedeutet, dass Vorhersage durchschnitt
    # um €500 vom tatsächlichen Wert abweicht

    return mae
```

5.3.2 RMSE (Root Mean Squared Error)

Formel: `sqrt(mean((y_true - y_pred)²))`

```
from sklearn.metrics import mean_squared_error
import numpy as np

def calculate_rmse(y_true, y_pred):
    """Root Mean Squared Error berechnen"""

    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    print(f"RMSE: €{rmse:,.2f}")

    # RMSE bestraft große Fehler stärker als MAE
    # Gut für: Ausreißer sind wichtig (z.B. sehr hohe Schäden)

    return rmse
```

5.3.3 R^2 Score (Determinationskoeffizient)

Formel: `1 - (SS_res / SS_tot)`

```
from sklearn.metrics import r2_score

def calculate_r2_score(y_true, y_pred):
    """R² Score berechnen"""

    r2 = r2_score(y_true, y_pred)
    print(f"R² Score: {r2:.4f}")

    # Interpretation:
    # 1.0 = Perfekte Vorhersage
    # 0.0 = Modell ist nicht besser als Mittelwert
    # < 0.0 = Modell ist schlechter als Mittelwert

    # In der Versicherung:
    # R² > 0.5 = Gutes Modell
    # R² > 0.7 = Sehr gutes Modell

    return r2
```

5.3.4 MAPE (Mean Absolute Percentage Error)

Formel: `mean(|y_true - y_pred| / y_true) × 100%`

```
def calculate_mape(y_true, y_pred):
    """Mean Absolute Percentage Error berechnen"""

    # Vermeide Division durch Null
    mask = y_true != 0
    mape = np.mean(np.abs((y_true[mask] - y_pred[mask]) / y_true[ma

    print(f"MAPE: {mape:.2f}%")

    # Interpretation: Durchschnittlicher prozentualer Fehler
    # Beispiel: MAPE von 15% bedeutet 15% Abweichung im Durchschnit

    return mape
```

5.3.5 Quantil-basierte Metriken

Wichtig für Versicherungen (z.B. VaR-Schätzung).

```
def calculate_quantile_metrics(y_true, y_pred, quantiles=[0.25, 0.5
    """Quantil-basierte Metriken"""

    results = {}

    for q in quantiles:
        # Tatsächliches Quantil
        true_quantile = np.quantile(y_true, q)

        # Vorhergesagtes Quantil
        pred_quantile = np.quantile(y_pred, q)

        # Fehler
        error = abs(true_quantile - pred_quantile)
        pct_error = (error / true_quantile) * 100 if true_quantile

        results[q] = {
            'true_quantile': true_quantile,
            'pred_quantile': pred_quantile,
            'error': error,
            'pct_error': pct_error
        }

        print(f"Quantil {q*100:.0f}%: "
              f"Tatsächlich €{true_quantile:,.2f}, "
              f"Vorhergesagt €{pred_quantile:,.2f}, "
              f"Fehler: {pct_error:.2f}%")

    return results
```

5.4 Cross-Validation

5.4.1 K-Fold Cross-Validation

Robuste Modellgüteprüfung.

```
from sklearn.model_selection import cross_val_score, KFold

def perform_cross_validation(model, X, y, cv=5, scoring='neg_mean_s
    """K-Fold Cross-Validation durchführen"""

    cv_scores = cross_val_score(model, X, y, cv=cv, scoring=scoring

    # Negate für MAE/RMSE (weil sklearn negative Scores zurückgibt)
    if 'neg' in scoring:
        cv_scores = -cv_scores

    print(f"Cross-Validation Scores ({cv}-fold):")
    print(f"  Mean: {cv_scores.mean():.4f}")
    print(f"  Std: {cv_scores.std():.4f}")
    print(f"  Min: {cv_scores.min():.4f}")
    print(f"  Max: {cv_scores.max():.4f}")

    return cv_scores
```

5.4.2 Time Series Cross-Validation

Für zeitliche Daten wichtig.

```
from sklearn.model_selection import TimeSeriesSplit

def perform_time_series_cv(model, X, y, n_splits=5):
    """Time Series Cross-Validation"""

    tscv = TimeSeriesSplit(n_splits=n_splits)
    scores = []

    for train_idx, test_idx in tscv.split(X):
        X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
        y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]

        model.fit(X_train, y_train)
        score = model.score(X_test, y_test)
        scores.append(score)

    print(f"Time Series CV Scores:")
    print(f"  Mean: {np.mean(scores):.4f}")
    print(f"  Std: {np.std(scores):.4f}")

    return scores
```

5.5 Residual-Analyse

```
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats

def analyze_residuals(y_true, y_pred):
    """Residual-Analyse durchführen"""

    residuals = y_true - y_pred

    fig, axes = plt.subplots(2, 2, figsize=(15, 12))

    # 1. Residual-Verteilung
    axes[0, 0].hist(residuals, bins=50, edgecolor='black')
    axes[0, 0].axvline(x=0, color='r', linestyle='--')
    axes[0, 0].set_title('Verteilung der Residuals')
    axes[0, 0].set_xlabel('Residual')

    # 2. Q-Q Plot (Normalverteilung prüfen)
    stats.probplot(residuals, dist="norm", plot=axes[0, 1])
    axes[0, 1].set_title('Q-Q Plot (Normalverteilung)')

    # 3. Residuals vs. Predicted
    axes[1, 0].scatter(y_pred, residuals, alpha=0.5)
    axes[1, 0].axhline(y=0, color='r', linestyle='--')
    axes[1, 0].set_xlabel('Vorhergesagte Werte')
    axes[1, 0].set_ylabel('Residuals')
    axes[1, 0].set_title('Residuals vs. Predicted')

    # 4. Residuals vs. Index (Zeit, falls vorhanden)
    axes[1, 1].plot(residuals)
    axes[1, 1].axhline(y=0, color='r', linestyle='--')
    axes[1, 1].set_xlabel('Index')
    axes[1, 1].set_ylabel('Residuals')
    axes[1, 1].set_title('Residuals über Zeit')

    plt.tight_layout()
    plt.show()
```

```
    # Statistische Tests
    # Shapiro-Wilk Test für Normalverteilung
    stat, p_value = stats.shapiro(residuals)
    print(f"\nShapiro-Wilk Test für Normalverteilung:")
    print(f"  Statistic: {stat:.4f}")
    print(f"  p-value: {p_value:.4f}")
    if p_value > 0.05:
        print("  Residuals sind normalverteilt ✓")
    else:
        print("  Residuals sind NICHT normalverteilt ✗")

    return residuals
```

5.6 Calibration (Kalibrierung)

```
from sklearn.calibration import calibration_curve, CalibratedClassi

def evaluate_calibration(y_true, y_pred_proba):
    """Calibration des Modells prüfen"""

    fraction_of_positives, mean_predicted_value = calibration_curve
        y_true, y_pred_proba, n_bins=10
    )

    plt.figure(figsize=(8, 6))
    plt.plot(mean_predicted_value, fraction_of_positives, "s-", lab
    plt.plot([0, 1], [0, 1], "k:", label="Perfekt kalibriert")
    plt.ylabel("Tatsächlicher Anteil Positiver")
    plt.xlabel("Vorhergesagte Wahrscheinlichkeit")
    plt.title("Calibration Plot")
    plt.legend()
    plt.grid(True)
    plt.show()

    # Calibration verbessern mit Platt Scaling oder Isotonic Regres
    calibrated_model = CalibratedClassifierCV(model, method='isotor
    calibrated_model.fit(X_train, y_train)

    return calibrated_model
```

5.7 Modellvergleich

```
def compare_models(models_dict, X, y):
    """Verschiedene Modelle vergleichen"""

    results = {}

    for name, model in models_dict.items():
        # Cross-Validation Scores
        cv_scores = cross_val_score(model, X, y, cv=5, scoring='neg

        results[name] = {
            'mean_cv_score': -cv_scores.mean(),
            'std_cv_score': cv_scores.std(),
            'min_cv_score': -cv_scores.max(),
            'max_cv_score': -cv_scores.min()
        }

        print(f"\n{name}:")
        print(f"  Mean CV Score: {results[name]['mean_cv_score']:.4
        print(f"  Std CV Score: {results[name]['std_cv_score']:.4f}

    # Bestes Modell finden
    best_model = min(results, key=lambda x: results[x]['mean_cv_sco
    print(f"\nBestes Modell: {best_model}")

    return results, best_model
```

5.8 Anwendungsfälle im operativen Controlling

- **Modellvalidierung**: Regelmäßige Prüfung der Modellgüte
- **Modellauswahl**: Vergleich verschiedener Ansätze
- **Monitoring**: Tracking der Modellperformance über Zeit
- **Reporting**: Dokumentation für Regulatoren/Audit

---

6. Operatives Controlling in der Sachversicherung

6.1 Aufgaben und Ziele

6.1.1 Hauptaufgaben

- **Schadenmonitoring**: Kontinuierliche Überwachung der Schadenentwicklung
- **Frühwarnsystem**: Identifikation von Abweichungen und Risiken
- **Performance-Messung**: Evaluation von Tarifen, Produkten, Segmenten
- **Reservierung**: Statistische Schadensreserven

6.1.2 Ziele

- **Kostensteuerung**: Verlustquoten kontrollieren
- **Risikofrüherkennung**: Trends rechtzeitig erkennen
- **Entscheidungsunterstützung**: Datenbasierte Empfehlungen
- **Compliance**: Regulatorische Anforderungen erfüllen

6.2 Typische Analysen

6.2.1 Schaden-Trend-Analyse

```
def analyze_claim_trends(df):
    """Schaden-Trends analysieren"""

    # Zeitliche Entwicklung
    monthly_trends = df.groupby(df['datum'].dt.to_period('M')).agg(
        'anzahl_schaeden': 'sum',
        'schadenshoehe': 'sum',
        'praemie': 'sum'
    })

    monthly_trends['loss_ratio'] = (
        monthly_trends['schadenshoehe'] / monthly_trends['praemie']
    ) * 100

    # Trend-Detection
    from scipy import stats

    # Linearer Trend testen
    x = np.arange(len(monthly_trends))
    slope, intercept, r_value, p_value, std_err = stats.linregress(
        x, monthly_trends['loss_ratio']
    )

    if p_value < 0.05:
        if slope > 0:
            print("⚠️ Signifikanter Anstieg in Loss Ratio!")
        else:
            print("✓ Loss Ratio sinkt signifikant")

    return monthly_trends
```

6.2.2 Segment-Performance

```
def analyze_segment_performance(df):
    """Performance nach Segmenten analysieren"""

    segment_perf = df.groupby('segment').agg({
        'praemie': 'sum',
        'schadenshoehe': 'sum',
        'anzahl_schaeden': 'sum',
        'versicherungssumme': 'sum'
    })

    segment_perf['loss_ratio'] = (
        segment_perf['schadenshoehe'] / segment_perf['praemie']
    ) * 100

    segment_perf['schadenquote'] = (
        segment_perf['anzahl_schaeden'] / len(df)
    ) * 100

    # Problemsegmente identifizieren
    problem_segments = segment_perf[segment_perf['loss_ratio'] > 10

    print("Problemsegmente (Loss Ratio > 100%):")
    print(problem_segments)

    return segment_perf
```

6.3 Reporting-Framework

```
def create_monthly_report(df, reporting_month):
    """Monatliches Controlling-Report erstellen"""

    report = {
        'reporting_date': reporting_month,
        'overview': {},
        'segments': {},
        'trends': {},
        'alerts': []
    }

    # Übersicht
    report['overview'] = {
        'total_premium': df['praemie'].sum(),
        'total_loss': df['schadenshoehe'].sum(),
        'loss_ratio': (df['schadenshoehe'].sum() / df['praemie'].su
        'total_claims': df['anzahl_schaeden'].sum(),
        'avg_claim_size': df['schadenshoehe'].sum() / df['anzahl_sc
    }

    # Alerts
    if report['overview']['loss_ratio'] > 100:
        report['alerts'].append({
            'type': 'CRITICAL',
            'message': f"Loss Ratio über 100%: {report['overview']['
        })

    if report['overview']['avg_claim_size'] > historical_avg * 1.2:
        report['alerts'].append({
            'type': 'WARNING',
            'message': f"Durchschnittliche Schadenhöhe um 20% über
        })

    return report
```

7. Case Study Vorbereitung

7.1 Typische Case Study Szenarien

7.1.1 Schadenfälle klassifizieren

Aufgabe: Entwickle ein Modell zur automatischen Klassifikation von Schadenfällen.

Lösungsansatz: 1. Daten explorieren (Schadentypen, Verteilungen) 2. Features engineer (historische Daten, Objekteigenschaften) 3. Verschiedene Klassifikationsmodelle testen (Logistic Regression, Random Forest, XGBoost) 4. Evaluation mit Confusion Matrix, F1-Score, ROC-AUC 5. Feature Importance analysieren 6. Deployment-Strategie überlegen

7.1.2 Loss Ratio Prognose

Aufgabe: Prognostiziere die Loss Ratio für das nächste Quartal.

Lösungsansatz: 1. Zeitreihenanalyse (Trends, Saisonalität) 2. Exogene Variablen identifizieren (Wetter, Marktdaten) 3. Time Series Modelle (ARIMA, Prophet, LSTM) 4. Regression-Modelle mit Features 5. Ensemble-Methoden 6. Backtesting

7.1.3 Tarifoptimierung

Aufgabe: Identifiziere Tarifgruppen mit Unter-/Übertarifierung.

Lösungsansatz: 1. Loss Ratio nach Tarifgruppen berechnen 2. Statistische Signifikanz testen 3. Erwartete vs. tatsächliche Schadenkosten vergleichen 4. Anpassungsempfehlungen entwickeln 5. Impact-Analyse durchführen

7.2 Wichtige Python-Skills für Case Study

```
# 1. Daten laden und explorieren
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv('schadensdaten.csv')
print(df.info())
print(df.describe())

# 2. Datenaufbereitung
# Fehlende Werte, Outlier, Feature Engineering

# 3. Modellierung
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier

# 4. Evaluation
from sklearn.metrics import classification_report, confusion_matrix

# 5. Visualisierung
# Wichtig für Präsentation!
```

7.3 Präsentationstipps

- **Struktur**: Problem → Lösung → Ergebnisse → Interpretation
- **Visualisierungen**: Klare, aussagekräftige Plots
- **Business Impact**: Verbindung zu Geschäftszielen
- **Limitations**: Ehrliche Diskussion von Einschränkungen
- **Next Steps**: Konkrete Handlungsempfehlungen

Wichtige Formeln - Quick Reference

Schadenkennzahlen

- **Loss Frequency Rate**: `Anzahl Schäden / Anzahl Versicherungsjahre`
- **Average Loss per Claim**: `Gesamtschaden / Anzahl Schäden`
- **Loss Ratio**: `(Gesamtschaden / Gesamtprämie) × 100%`

Modellgüte - Klassifikation

- **Accuracy**: `(TP + TN) / (TP + TN + FP + FN)`
- **Precision**: `TP / (TP + FP)`
- **Recall**: `TP / (TP + FN)`
- **F1-Score**: `2 × (Precision × Recall) / (Precision + Recall)`

Modellgüte - Regression

- **MAE**: `mean(|y_true - y_pred|)`
- **RMSE**: `sqrt(mean((y_true - y_pred)²))`
- **R^2**: `1 - (SS_res / SS_tot)`

Wahrscheinlichkeitsverteilungen

- **Poisson**: $P(k) = (\lambda^k \times e^{-\lambda}) / k!$
- **Lognormal**: Parametrisiert durch μ, σ

Wichtige Python-Bibliotheken

```
# Datenmanipulation
import pandas as pd
import numpy as np

# Machine Learning
from sklearn.model_selection import train_test_split, cross_val_scc
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.ensemble import RandomForestClassifier, GradientBoosti
from sklearn.linear_model import LogisticRegression, PoissonRegress
from sklearn.metrics import (accuracy_score, precision_score, recal
                             f1_score, roc_auc_score, mean_absolute_
                             mean_squared_error, r2_score)

# Statistik
from scipy import stats
import statsmodels.api as sm
from statsmodels.genmod import families

# Visualisierung
import matplotlib.pyplot as plt
import seaborn as sns

# Zeitreihen
from statsmodels.tsa.arima.model import ARIMA
from prophet import Prophet

# Moderne ML
import xgboost as xgb
from lightgbm import LGBMClassifier
```

Checkliste für Vorstellungsgespräch

- [ ] Verstehe die vier Hauptthemen (Schadenklassifikation, Wahrscheinlichkeitsmodelle, Tarifierung, Kennzahlensystem)
- [ ] Kenne die wichtigsten Metriken zur Modellgüteprüfung
- [ ] Kann Python-Code für versicherungsspezifische Analysen schreiben
- [ ] Verstehe Loss Ratio, Claim Frequency, Claim Severity
- [ ] Kenne GLM, Poisson Regression, Gamma Regression
- [ ] Kann Confusion Matrix, ROC-AUC, F1-Score interpretieren
- [ ] Verstehe die Rolle des operativen Controllings
- [ ] Bereit für Case Study mit strukturiertem Ansatz

Dieses Dokument dient als umfassende Vorbereitung für das Vorstellungsgespräch als Data Scientist im operativen Controlling der Sachversicherung.

Viel Erfolg bei deinem Vorstellungsgespräch! 🎯